ECE637 Digital Image Processing I
Laboratory work 1:
Image Filtering

Boris Diner

January 28, 2021

3 FIR Low Pass Filter

3.1 The analytical expression for $H(e^{j\mu}, e^{j\nu})$

Given a low pass filter $h(m, n)$ such that

$$h(m,n) = \begin{cases} \frac{1}{81}, & \text{for } |m| \leq 4, |n| \leq 4 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

By definition, the DSFT of $h(m, n)$ is

$$H(e^{j\mu}, e^{j\nu}) \triangleq \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} h(m,n) \cdot e^{-j(\mu m + \nu n)} \tag{2}$$

Plugging expression 1 in 2, we get

$$H(e^{j\mu}, e^{j\nu}) = \frac{1}{81} \cdot \sum_{m=-4}^{4} e^{-j\mu m} \sum_{n=-4}^{4} e^{-j\nu n} \tag{3}$$

In general, for any arbitrary natural number N a sum of complex exponentials can be rewritten as

$$\sum_{m=-N}^{N} e^{-j\mu m} = \sum_{m'=0}^{2N} e^{-j\mu(m'-N)} = e^{j\mu N} \cdot \sum_{m'=0}^{2N} e^{-j\mu m'} \tag{4}$$

The sum of the first $2N+1$ terms of the geometric sequence of complex exponentials is equal to

$$\sum_{m'=0}^{2N} e^{-j\mu m'} = \frac{1-e^{-j\mu(2N+1)}}{1-e^{-j\mu}} \tag{5}$$

Combining 4 with 5 and performing some operations, we obtain the following result

$$e^{j\mu N} \cdot \frac{1-e^{-j\mu(2N+1)}}{1-e^{-j\mu}} = e^{j\mu N} \cdot \frac{e^{-j\mu(2N+1)/2}}{e^{-j\mu/2}} \cdot \frac{e^{j\mu(2N+1)/2}-e^{-j\mu(2N+1)/2}}{e^{j\mu/2}-e^{-j\mu/2}} \tag{6}$$

It can be derived from Euler's formula that

$$\sin x = \frac{e^{jx}-e^{-jx}}{2j} \tag{7}$$

Applying 7 to 6, we get

$$\sum_{m=-N}^{N} e^{-j\mu m} = \frac{\sin\left(\frac{(2N+1)\mu}{2}\right)}{\sin\left(\frac{\mu}{2}\right)} \tag{8}$$

Using the fact that we are interested in the case of $N = 4$ and generalizing the result for 2-D case, we find the analytical expression for $H(e^{j\mu}, e^{j\nu})$

$$H(e^{j\mu}, e^{j\nu}) = \frac{1}{81} \cdot \frac{\sin\frac{9\mu}{2} \cdot \sin\frac{9\nu}{2}}{\sin\frac{\mu}{2} \cdot \sin\frac{\nu}{2}} \tag{9}$$

Therefore, the magnitude of the frequency response is

$$|H(e^{j\mu}, e^{j\nu})| = \frac{1}{81} \cdot \left|\frac{\sin\frac{9\mu}{2} \cdot \sin\frac{9\nu}{2}}{\sin\frac{\mu}{2} \cdot \sin\frac{\nu}{2}}\right|$$

3.2 The plot of $|H(e^{j\mu}, e^{j\nu})|$

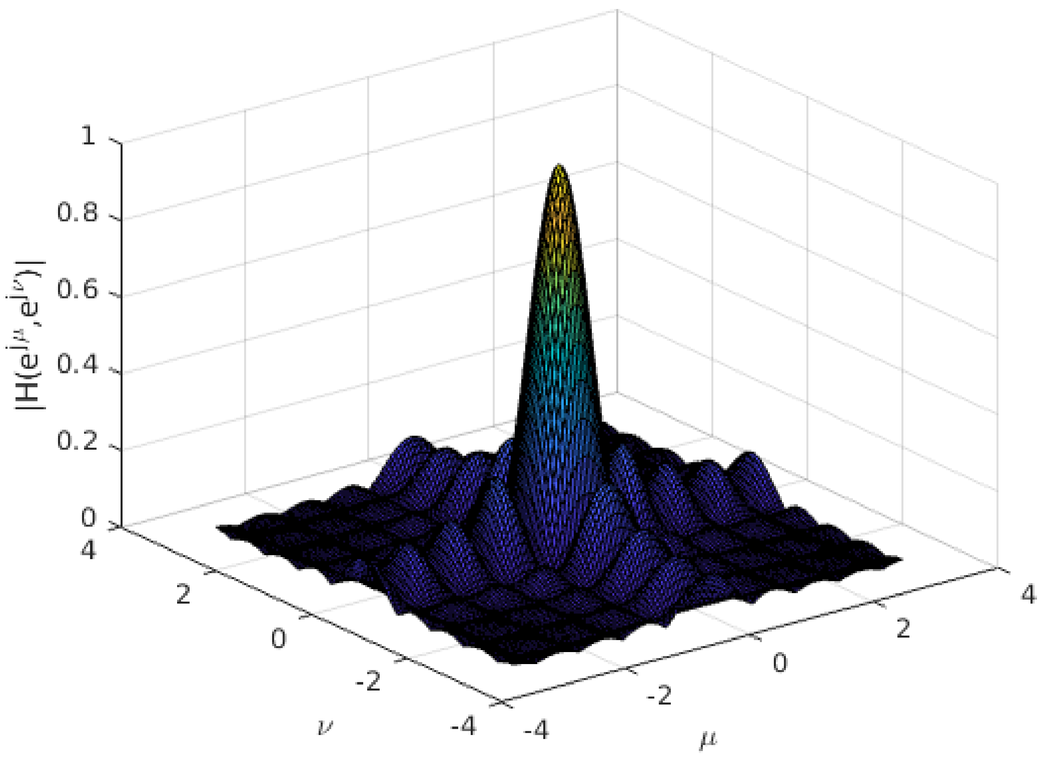


Figure 1: The magnitude of the frequency response $|H(e^{j\mu}, e^{j\nu})|$

3.3 The color image and the filtered image

(a) img03.tif (b) img-lowpass.tif

Figure 2: The result of filtering the original image (a) with the low-pass filter $H(e^{j\mu}, e^{j\nu})$

3.4 C code listing for filtering the image

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

#define FILTER_WIDTH 9
#define FILTER_HEIGHT 9

void error(char *name);
double crop_image(double pixel);

int main (int argc, char **argv)
```

```
{
FILE *fp;
  struct TIFF_img input_img, color_img;
  double **img_temp;
  int i,j,k,l;
  uint8_t channel;
  double sum;
  double h = 1.0 /  (FILTER_WIDTH * FILTER_HEIGHT);
  int half_filter_width = (FILTER_WIDTH - 1) / 2;
  int half_filter_height = (FILTER_HEIGHT - 1) / 2;


  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 != 'c' ) {
    fprintf ( stderr, "error: image must be 24-bit color\n" );
    exit ( 1 );
  }

```

```
get_TIFF ( &color_img, input_img.height, input_img.width , 'c' );

/* Filter each of three color channels of the image */
for (channel = 0; channel < 3; channel++){

  img_temp = (double**)get_img(input_img.width + (FILTER_WIDTH - 1), input_img.height + (FILTER_HEIGHT - 1),sizeof(double)); // Allocate image of double precision floats

  /* Free boundaries */
  for(i = 0; i < input_img.height + (FILTER_HEIGHT - 1); i++){
    for(j = 0; j < input_img.width + (FILTER_WIDTH - 1); j++){
      if ((i >= half_filter_height && i < (input_img.height + half_filter_height)) && (j >= half_filter_width && ( j < input_img.width + half_filter_width))){
        img_temp[i][j] = input_img.color[channel][i-half_filter_height][j-half_filter_width];
      }
      else {
        img_temp[i][j] = 0;
      }
    }
  }
  /* Filtering process */
  for ( i = 0; i < input_img.height; i++ ){
    for ( j = 0; j < input_img.width; j++ ){
      sum = 0.0;
      for (k = 0; k < FILTER_HEIGHT; k++ ){
        for (l = 0; l < FILTER_WIDTH; l++ ){
          sum += h * img_temp[i+k][j+l];
        }
```

```
        }
        color_img.color[channel][i][j] = crop_image(sum);
      }
    }

    free_img((void**)img_temp); // Clear allocated memory after each iteration

  }

  /* open color image file */
  if ( ( fp = fopen ( "img-lowpass.tif", "wb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file color.tif\n");
    exit ( 1 );
  }

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

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

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

  return(0);
}

void error(char *name)
```

```
{
  printf("usage: %s image.tiff \n\n",name);
  printf("this program reads in a 24-bit color TIFF image
   .\n");
  printf("It then horizontally filters the green component
   , adds noise,\n");
  printf("and writes out the result as an 8-bit image\n");
  printf("with the name 'green.tiff'.\n");
  printf("It also generates an 8-bit color image,\n");
  printf("that swaps red and green components from the
   input image");
  exit(1);
}
double crop_image(double pixel)
{
  if (pixel > 255) return 255;
  if (pixel < 0)  return 0;
  return pixel;
}
```

4 FIR Sharpening Filter

4.1 The analytical expression for $H(e^{j\mu}, e^{j\nu})$

Using the method described in Section 3 for $N = 2$ and for the normalization factor of $\frac{1}{25}$), we get

$$H(e^{j\mu}, e^{j\nu}) = \frac{1}{25} \cdot \frac{\sin\frac{5\mu}{2} \cdot \sin\frac{5\nu}{2}}{\sin\frac{\mu}{2} \cdot \sin\frac{\nu}{2}} \tag{10}$$

4.2 The analytical expression for $G(e^{j\mu}, e^{j\nu})$

Taking into account the fact that

$$\text{DSFT}\{\delta(m,n)\} = \text{DTFT}\{\delta(m)\} \cdot \text{DTFT}\{\delta(n)\} = 1$$

and using the linearity property of the DTFT, we get

$$G(e^{j\mu}, e^{j\nu}) = 1 + \lambda[1 - H(e^{j\mu}, e^{j\nu})] \tag{11}$$

Plugging 10 in 11, we derive the analytical expression for the sharpening filter $G(e^{j\mu}, e^{j\nu})$

$$G(e^{j\mu}, e^{j\nu}) = 1 + \lambda \left(1 - \frac{1}{25} \cdot \frac{\sin\frac{5\mu}{2} \cdot \sin\frac{5\nu}{2}}{\sin\frac{\mu}{2} \cdot \sin\frac{\nu}{2}}\right)$$

As $\text{Im}\{G\} = 0$, the magnitude of the frequency response is

$$|G(e^{j\mu}, e^{j\nu})| = \left|1 + \lambda \left(1 - \frac{1}{25} \cdot \frac{\sin\frac{5\mu}{2} \cdot \sin\frac{5\nu}{2}}{\sin\frac{\mu}{2} \cdot \sin\frac{\nu}{2}}\right)\right|$$

4.3 The plot of $|H(e^{j\mu}, e^{j\nu})|$

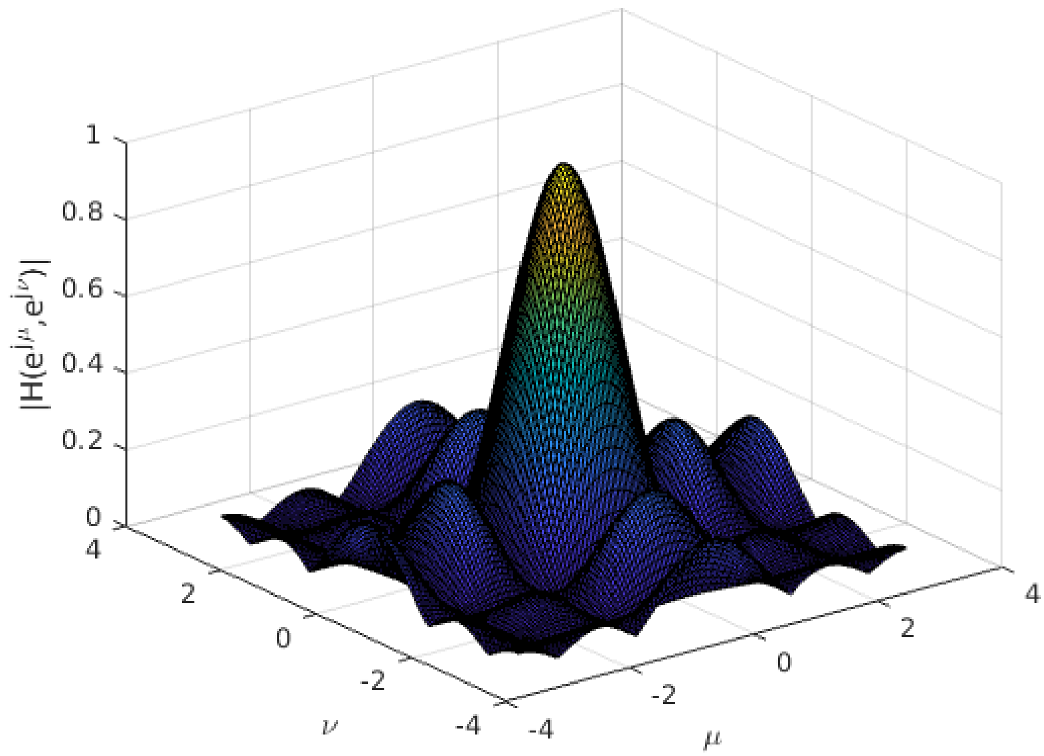


Figure 3: The magnitude of the frequency response $|H(e^{j\mu}, e^{j\nu})|$

4.4 The plot of $|G(e^{j\mu}, e^{j\nu})|$ for $\lambda = 1.5$

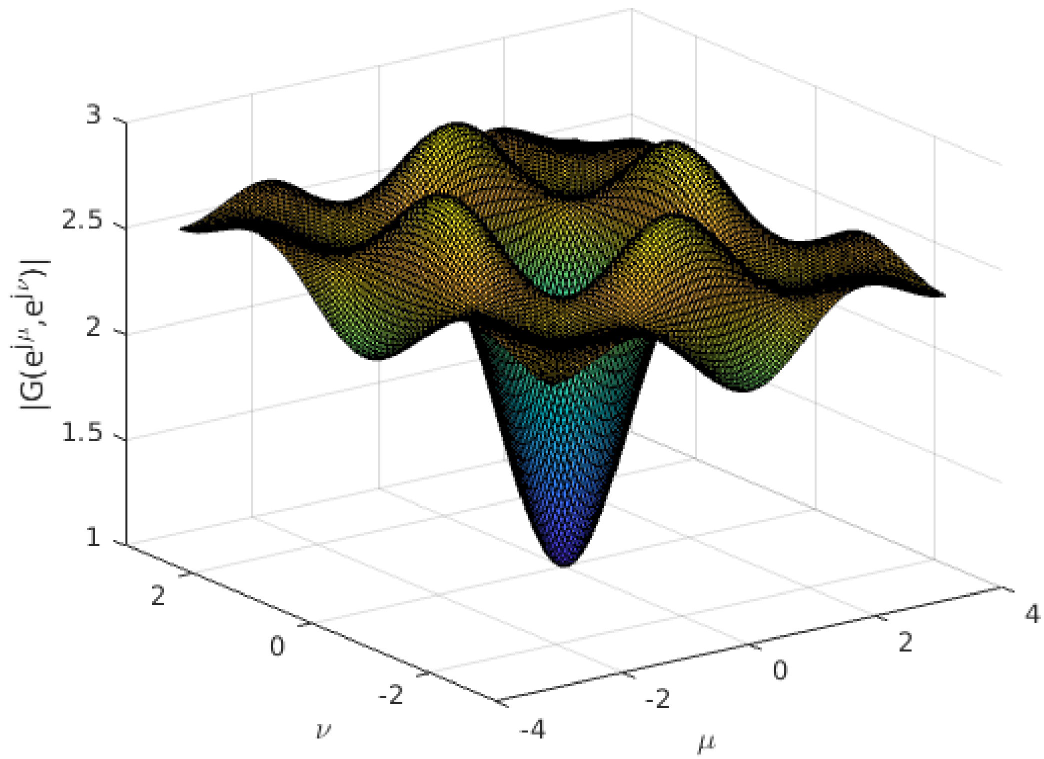
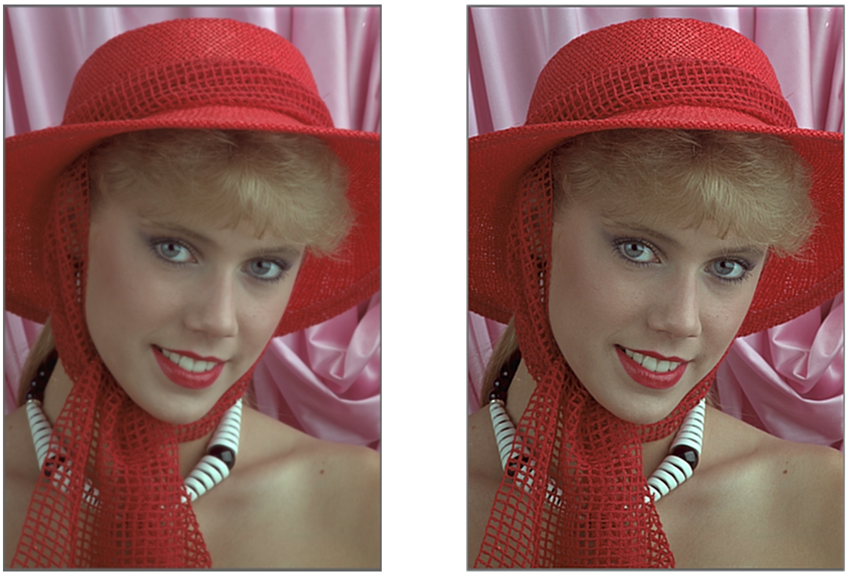


Figure 4: The magnitude of the frequency response $|G(e^{j\mu}, e^{j\nu})|$ for $\lambda = 1.5$

4.5 The color image and the sharpened image



(a) imgblur.tif (b) img-highpass.tif

Figure 5: The result of sharpening the image (a) with the high-pass filter $G(e^{j\mu}, e^{j\nu})$

4.6 C code listing for filtering the image

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

#define FILTER_WIDTH 5
#define FILTER_HEIGHT 5

void error(char *name);
double crop_image(double pixel);

int main (int argc, char **argv)
```

```
{
FILE *fp;
  struct TIFF_img input_img, color_img;
  double **img_temp;
  int i,j,k,l;
  uint8_t channel;
  double sum;
  double h = 1.0 / (FILTER_WIDTH * FILTER_HEIGHT);
  double lambda = 1.5;
  double delta;
  double sharpening_filter[FILTER_WIDTH][FILTER_HEIGHT];
  int half_filter_width = (FILTER_WIDTH - 1) / 2;
  int half_filter_height = (FILTER_HEIGHT - 1) / 2;


  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 != 'c' ) {
    fprintf ( stderr, "error: image must be 24-bit color\n" );
```

```
    exit ( 1 );
  }

  get_TIFF ( &color_img, input_img.height, input_img.width, 'c' );

  /* Fill in the sharpening filter */
  for (i = 0; i < FILTER_WIDTH; i++) {
    for(j = 0; j < FILTER_HEIGHT; j++) {
        if(i == half_filter_width && j == half_filter_height) {
            delta = 1.0; // the center of the filter
        }
        else {
            delta = 0.0;
        }
    sharpening_filter[i][j] = delta + lambda * (delta - h);
    }
  }

  /* Filter each of three color channels of the image */
  for (channel = 0; channel < 3; channel++){

    img_temp = (double**)get_img(input_img.width + (FILTER_WIDTH - 1), input_img.height + (FILTER_HEIGHT - 1),sizeof(double)); // Allocate image of double precision floats

    /* Free boundaries */
    for(i = 0; i < input_img.height + (FILTER_HEIGHT - 1); i++){
      for(j = 0; j < input_img.width + (FILTER_WIDTH - 1); j++){
        if ((i >= half_filter_height && i < (input_img.
```

```
height + half_filter_height)) && (j >= half_filter_width
 && ( j < input_img.width + half_filter_width))){
          img_temp[i][j] = input_img.color[channel][i-
half_filter_height][j-half_filter_width];
        }
        else {
          img_temp[i][j] = 0;
        }
      }
    }
    /* Filtering process */
    for ( i = 0; i < input_img.height; i++ ){
      for ( j = 0; j < input_img.width; j++ ){
        sum = 0.0;
        for (k = 0; k < FILTER_HEIGHT; k++ ){
          for (l = 0; l < FILTER_WIDTH; l++ ){
            sum += sharpening_filter[k][l] * img_temp[i+k
][j+l];
          }
        }
        color_img.color[channel][i][j] = crop_image(sum);
      }
    }

    free_img((void**)img_temp); // Clear allocated memory
 after each iteration

  }

  /* open color image file */
  if ( ( fp = fopen ( "img-highpass.tif", "wb" ) ) == NULL
   ) {
    fprintf ( stderr, "cannot open file color.tif\n");
    exit ( 1 );
  }
```

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

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

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

  return(0);
}

void error(char *name)
{
  printf("usage: %s image.tiff \n\n",name);
  printf("this program reads in a 24-bit color TIFF image.\n");
  printf("It then horizontally filters the green component, adds noise,\n");
  printf("and writes out the result as an 8-bit image\n");
  printf("with the name 'green.tiff'.\n");
  printf("It also generates an 8-bit color image,\n");
  printf("that swaps red and green components from the input image");
  exit(1);
}
double crop_image(double pixel)
{
  if (pixel > 255) return 255;
```

```
  if (pixel < 0)  return 0;
  return pixel;
}
```

5 IIR Filter

5.1 The analytical expression for $H(e^{j\mu}, e^{j\nu})$

Given the 2-D difference equation

$$\begin{aligned} y(m,n) = 0.01x(m,n) &+ 0.9y(m-1,n) \\ &+ 0.9y(m,n-1) - 0.81y(m-1,n-1) \end{aligned} \tag{12}$$

where $x(m,n)$ is the input and $y(m,n)$ is the output.

Let $h(m,n)$ be the impulse response of an IIR filter with the corresponding difference equation.

Applying Z-transform to this expression, we get

$$\begin{aligned} Y(z_1,z_2) = 0.01X(z_1,z_2) &+ 0.9z_1^{-1}Y(z_1,z_2) \\ &+ 0.9z_2^{-1}Y(z_1,z_2) - 0.81z_1^{-1}z_2^{-1}Y(z_1,z_2) \end{aligned}$$

Then the DSFT of the impulse response $h(m,n)$ is equal to

$$\begin{aligned} H(z_1,z_2)\Big|_{\substack{z_1=\exp(j\mu)\\ z_2=\exp(j\nu)}} &= \frac{Y(z_1,z_2)}{X(z_1,z_2)}\Bigg|_{\substack{z_1=\exp(j\mu)\\ z_2=\exp(j\nu)}} = \\ &= \frac{0.01}{1-0.9e^{-j\mu}-0.9e^{-j\nu}+0.81e^{-j\mu}e^{-j\nu}} \end{aligned}$$

5.2 The plot of $|H(e^{j\mu}, e^{j\nu})|$

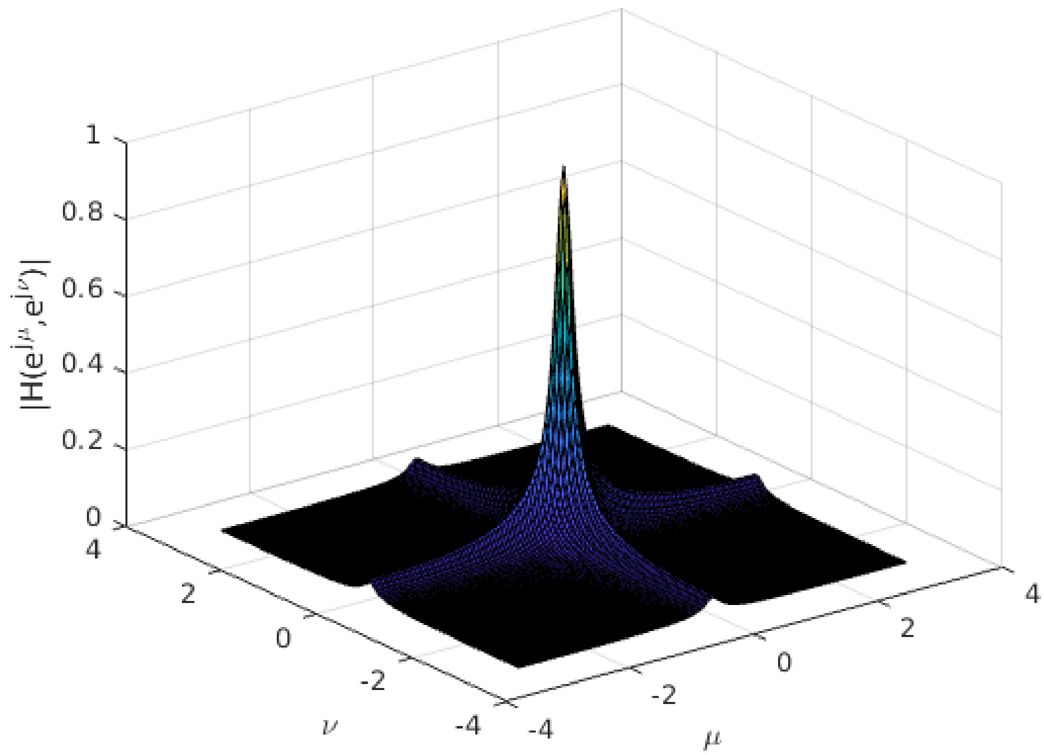


Figure 6: The magnitude of the frequency response $|H(e^{j\mu}, e^{j\nu})|$

5.3 The image of the point spread function

Applying the difference equation to a 256×256 image of the form $x(m, n) = \delta(m - 127, n - 127)$, we get the following result.

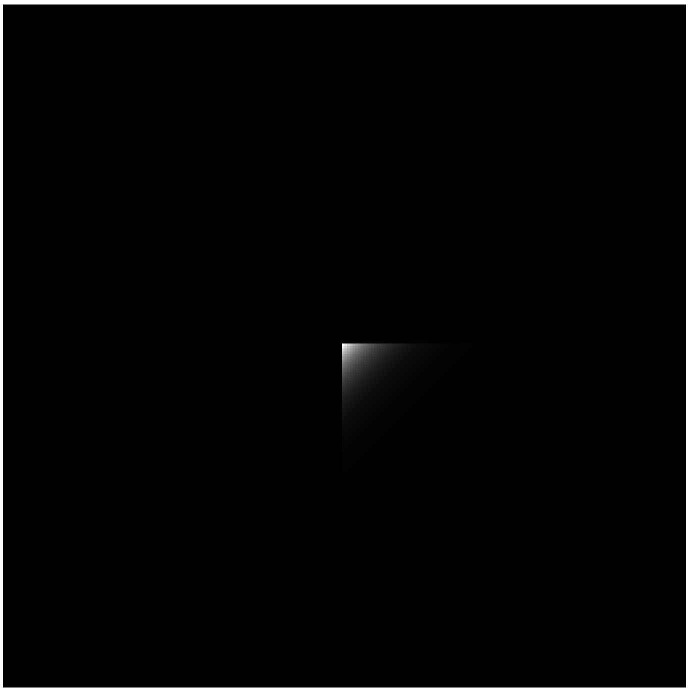


Figure 7: The image of the point spread function

5.4 The filtered output color image

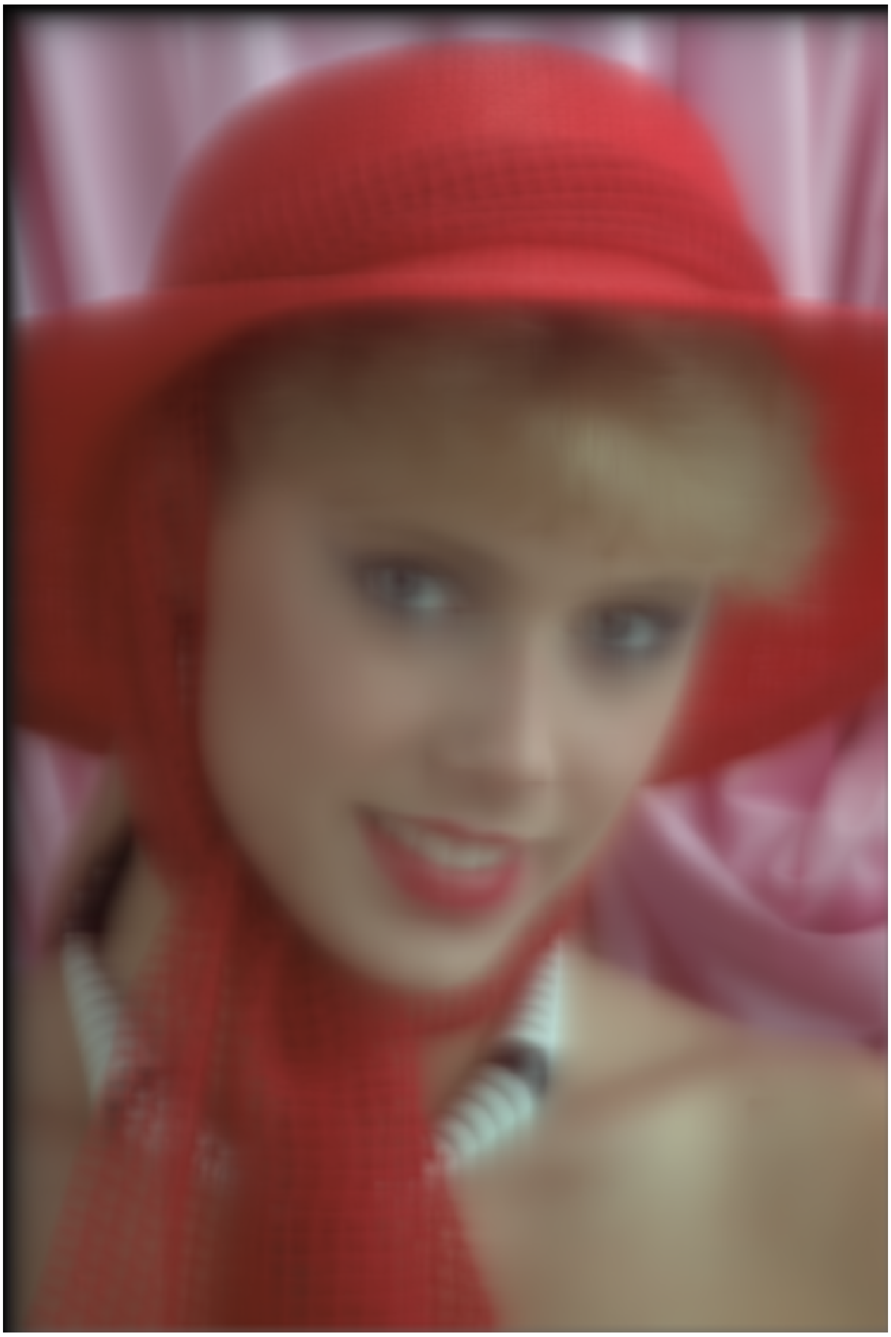


Figure 8: The filtered output image img-iir.tif

5.5 C code listing for filtering the image

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

void error(char *name);
double crop_image(double pixel);

int main (int argc, char **argv)
{
FILE *fp;
  struct TIFF_img input_img, color_img;
  double **img_temp;
  int i,j,k,l;
  uint8_t channel;

  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 != 'c' ) {
  fprintf ( stderr, "error: image must be 24-bit color\n" );
  exit ( 1 );
}

get_TIFF ( &color_img, input_img.height, input_img.width, 'c' );

/* Filter each of three color channels of the image */
for (channel = 0; channel < 3; channel++){

  img_temp = (double**)get_img(input_img.width + 1, input_img.height + 1,sizeof(double)); // Allocate image of double precision floats

  for(i = 0;i < input_img.height + 1; i++){
    for(j = 0;j < input_img.width + 1; j++){
    img_temp[i][j] = 0;
    }
  }
  for (k = 0; k < input_img.height; k++){
    for (l = 0; l < input_img.width; l++){
      img_temp[k+1][l+1] = 0.01*input_img.color[channel][k][l]+0.9*img_temp[k][l+1]+0.9*img_temp[k+1][l]-0.81*img_temp[k][l];
      color_img.color[channel][k][l] = crop_image(img_temp[k+1][l+1]);
    }
  }
  free_img( (void**)img_temp); // Clear allocated memory after each iteration
}

```

```
  /* open color image file */
  if ( ( fp = fopen ( "img-iir.tif", "wb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file color.tif\n");
    exit ( 1 );
  }

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

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

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

  return(0);
}

void error(char *name)
{
  printf("usage: %s image.tiff \n\n",name);
  printf("this program reads in a 24-bit color TIFF image
   .\n");
  printf("It then horizontally filters the green component
   , adds noise,\n");
  printf("and writes out the result as an 8-bit image\n");
  printf("with the name 'green.tiff'.\n");
  printf("It also generates an 8-bit color image,\n");
  printf("that swaps red and green components from the
   input image");
```

```
  exit(1);
}

double crop_image(double pixel)
{
  if (pixel > 255) return 255;
  if (pixel < 0)  return 0;
  return pixel;
}
```