Report Lab 1 – Misha Tsysin, 0033922418

# FIR Low Pass Filter

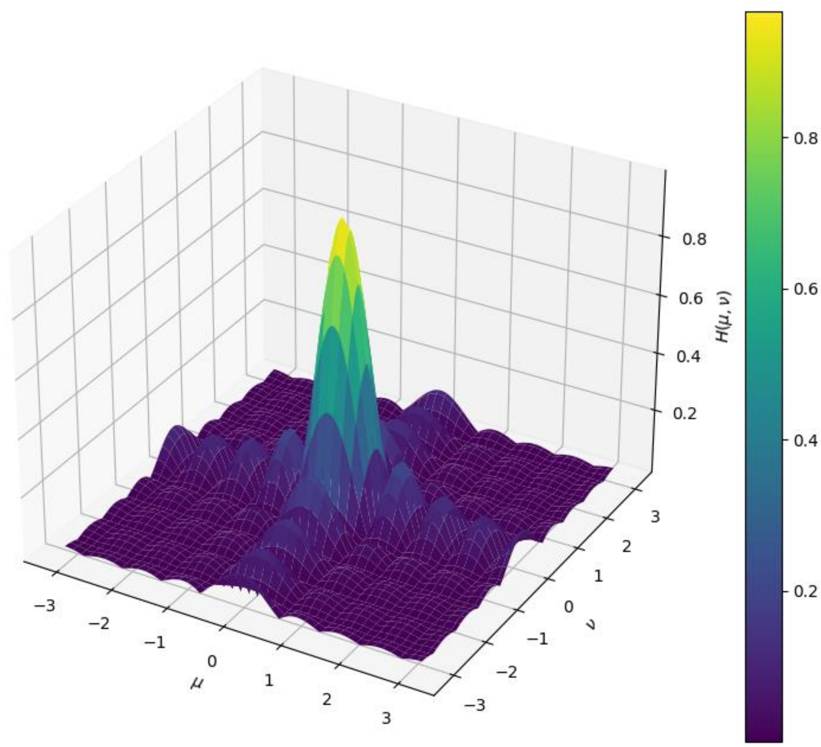
Let's derive the equation for $H(e^{j\mu}, e^{j\nu})$. The original filter has the expression:

$$h(m,n) = rect\left(\frac{m}{9}\right) rect\left(\frac{n}{9}\right) * \frac{1}{81} = \left(\frac{1}{9} rect\left(\frac{m}{9}\right)\right)\left(\frac{1}{9} rect\left(\frac{n}{9}\right)\right)$$

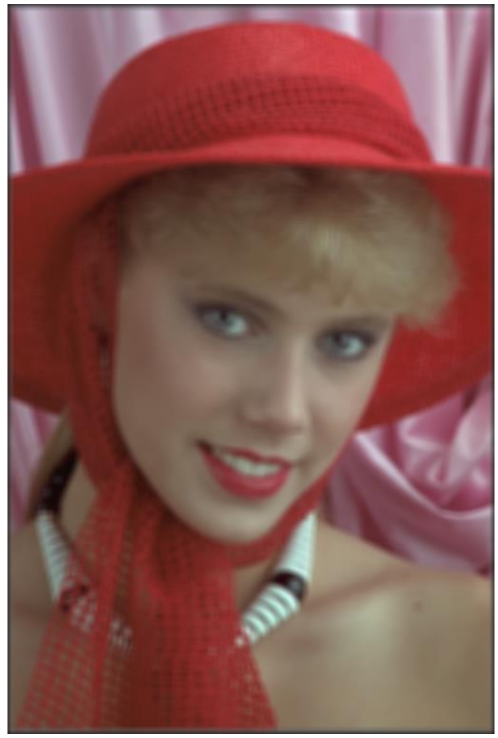
From the separability property and the standard result from the notes ($\mathcal{F}\left(rect\left(\frac{n}{N}\right)\right) = \frac{sin(n\omega/2)}{sin(\omega/2}$) we get:

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

The magnitude of the DSFT will be look like:



The original and filtered images look like this:



The code used for this part is embedded in the common code.

The lines that correspond to filter generation and application are:

```
    allocate_img3(&filter, 9, 9);
    for ( i = 0; i < filter.height; i++ )
    for ( j = 0; j < filter.width; j++ )
    for (int ch = 0; ch < 3; ch++) {
      filter.img[ch][i][j] = 1.0/81.0;
    }
    apply_filter_3(input_img, filter, filtered_image);
```

# FIR Sharpening Filter

Similarly to the previous part, let's derive a transfer function:

The original filter $h(m,n)$ has the following definition:

$$h(m,n) = rect\left(\frac{m}{5}\right) rect\left(\frac{n}{5}\right) * \frac{1}{25}$$

This looks exactly like the filter in the previous part! Thus, the expression for $H(e^{j\mu}, e^{j\nu})$ will be:

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

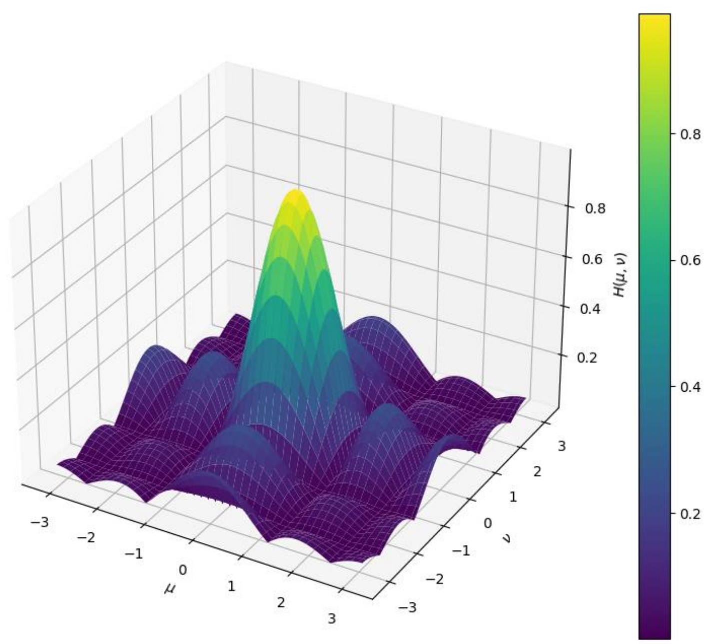
The actual sharpening filter has the following form:

$$g(m,n) = \delta(m,n) + \lambda(\delta(m,n) - h(m,n))$$

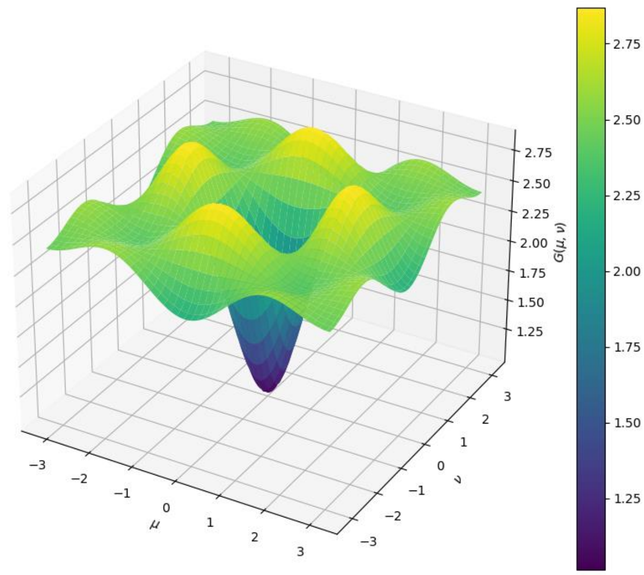
Thus, using the fact that $\mathcal{F}(\delta(m,n)) = 1$:

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

The magnitude of the DSFT will look like for $H(e^{j\mu}, e^{j\nu})$:



And for $G\left(e^{j\mu}, e^{j\nu}\right)$:




The blurred and filtered images look like this:

The code snippet corresponding to generating the image is:

```
lambda = atof(argv[4]);
printf("Lambda:  %f \n", lambda);

allocate_img3(&filter, 5, 5);

int fcenter_y =  filter.height / 2;
int fcenter_x =  filter.width / 2;
for ( i = 0; i < filter.height; i++ )
for ( j = 0; j < filter.width; j++ ) {
  int delta  = (int)(i==fcenter_y && j==fcenter_x);
  for (int ch = 0; ch < 3; ch++) {
    filter.img[ch][i][j] = delta + lambda * (delta - 1.0/25.0);
  }
}
apply_filter_3(input_img, filter, filtered_image);
```

# IIR Filter

We have the following difference equation:

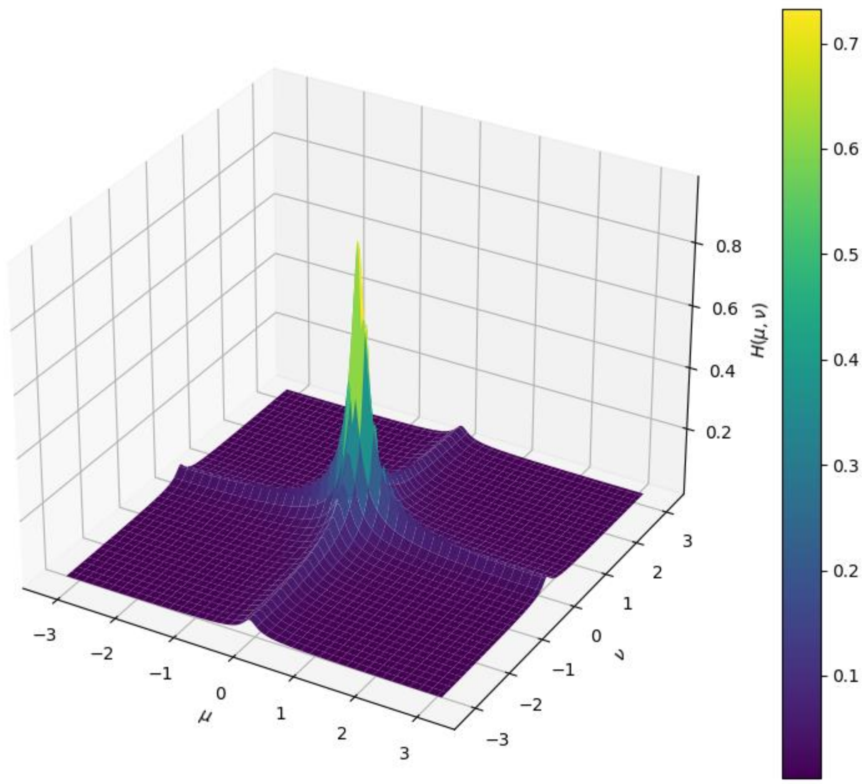
$$y(m,n) = 0.01x(m,n) + 0.9\big(y(m-1,n) + y(m,n-1)\big) - 0.81y(m-1,n-1)$$

Take the Fourier transform of both sides:

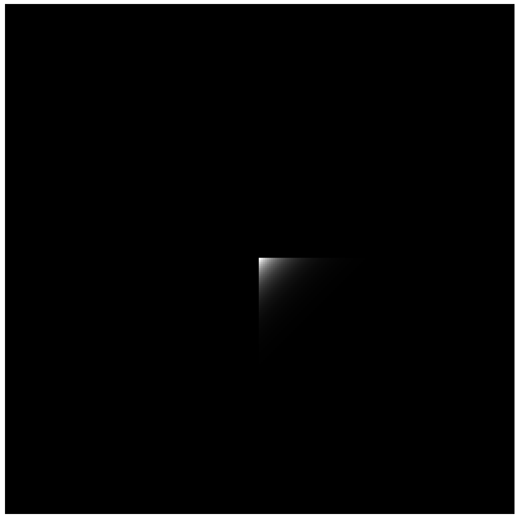
$$\begin{aligned} Y\left(e^{j\mu}, e^{j\nu}\right) = {} & 0.01X\left(e^{j\mu}, e^{j\nu}\right) + 0.9\left(Y\left(e^{j\mu}, e^{j\nu}\right)e^{-j\mu} + Y\left(e^{j\mu}, e^{j\nu}\right)e^{-j\nu}\right) \\ & - 0.81yY\left(e^{j\mu}, e^{j\nu}\right)e^{-j(\mu+\nu)} \end{aligned}$$

$$H\left(e^{j\mu}, e^{j\nu}\right) = \frac{Y\left(e^{j\mu}, e^{j\nu}\right)}{X\left(e^{j\mu}, e^{j\nu}\right)} = \frac{0.01}{(1 - 0.9e^{-j\mu})(1 - 0.9e^{-j\nu})} = \frac{0.01}{1 - 0.9e^{-j\mu} - 0.9e^{-j\nu} + 0.81e^{-j(\mu+\nu)}}$$

The magnitude of this FIR filter will look like:



The point spread function will look like:



The original / processed images are

The code snippet corresponding to generating the image is:

```
    // NOTE:::
    // In tiff.c certain flag checks were removed (XResolution and YResolution) Line 2366 -
2367
    // Slow version (not running)
    if (0) {
      open_routine(fp, argv[4], &filter_tiff, 'g');
      allocate_img3(&filter_generated, filter_tiff.height, filter_tiff.width);

      for ( i = 0; i < filter_generated.height; i++ )
      for ( j = 0; j < filter_generated.width; j++ )
      for (int ch = 0; ch < 3; ch++) {
        filter_generated.img[ch][i][j] = filter_tiff.mono[i][j];
      }

      apply_filter_3(input_img, filter_generated, filtered_image);
    } else {
      // Fast version, just plainly apply the recursive relation:
      for ( i = 0; i < filtered_image.height; i++ )
      for ( j = 0; j < filtered_image.width; j++ )
      for (int ch = 0; ch < 3; ch++) {
        filtered_image.img[ch][i][j] = 0.01*input_img.img[ch][i][j];
        if (i) filtered_image.img[ch][i][j] += 0.9*filtered_image.img[ch][i-1][j];
        if (j) filtered_image.img[ch][i][j] += 0.9*filtered_image.img[ch][i][j-1];
        if (i&&j) filtered_image.img[ch][i][j] -= 0.81*filtered_image.img[ch][i-1][j-1];
      }
    }
```

## Full Code:

Here is the full code for the lab. I decided to reuse as much code as possible between the tasks so it's hard to separate code between them. The main functions that apply filters are apply_filter() and apply_filter_3() for mono and color images respectively.

filter.h

```
#ifndef _FILTER_H_
#define _FILTER_H_

#include <math.h>
#include "tiff.h"
#include "allocate.h"
#include <assert.h>

typedef struct {
  int32_t     height;
  int32_t     width;
  double**    img;
} image_t;

typedef struct {
  int32_t     height;
  int32_t     width;
  double** img[3];
} image3_t;

void free_image(image_t image);
void free_image_3(image3_t image);

void error(char *name);
void open_routine(FILE *fp, char* filename, struct TIFF_img* read_image, char type_check);
void write_routine(FILE *fp, char* filename, struct TIFF_img* write_image);
void allocte_img3( struct TIFF_img* tiff_source_image, image3_t* target_img);
void populate_tiff_from_img3(image3_t* img, struct TIFF_img* color_img);


void apply_filter(image_t input, image_t filter, image_t output);
void apply_filter_3(image3_t input, image3_t filter, image3_t output);

#endif //_FILTER_H_
```

filter.c

```c
#include "filter.h"

void free_image(image_t image) {
  free_img((void**) image.img);
}
void free_image_3(image3_t image) {
  for (int ch = 0; ch < 3; ch++) free_img((void**) image.img[ch]);
}

void apply_filter(image_t input, image_t filter, image_t output) {
  // Applies filter to img all on tpre-allocated memory
  // Pretty inefficient approach with a bunch of for-loops
  assert(input.height == output.height);
  assert(input.width == output.width);
  assert(filter.height % 2);
  assert(filter.width % 2);

  int32_t shift_h = (filter.height - 1) / 2;
  int32_t shift_w = (filter.width - 1) / 2;

  for (int32_t i = 0; i < input.height; i++)
  for (int32_t j = 0; j < input.width; j++) {
    output.img[i][j] = 0.0;
    // fprintf ( stderr, "sdfdf %i, %i\n", i, j );
    for (int32_t ii = 0; ii < filter.height; ii++)
    for (int32_t jj = 0; jj < filter.width; jj++) {
      int idx_im_y = i - ii + shift_h;
      int idx_im_x = j - jj + shift_w;
      int idx_f_y = ii;
      int idx_f_x = jj;
      if (0 <= idx_im_y && idx_im_y < input.height &&
      0 <= idx_im_x && idx_im_x < input.width &&
      0 <= idx_f_y && idx_f_y < filter.height &&
      0 <= idx_f_x && idx_f_x < filter.width) {
        output.img[i][j] += input.img[idx_im_y][idx_im_x] *
                            filter.img[idx_f_y][idx_f_x];
      }
    }
  }
}

void apply_filter_3(image3_t input, image3_t filter, image3_t output) {
  for (int ch = 0; ch < 3; ch++) {
```

```c
    image_t channel_img = {input.height, input.width, input.img[ch]};
    image_t channel_filter = {filter.height, filter.width, filter.img[ch]};
    image_t channel_output = {output.height, output.width, output.img[ch]};
    apply_filter(channel_img, channel_filter, channel_output);
  }
}

void open_routine(FILE *fp, char* filename, struct TIFF_img* read_image, char type_check) {
  /* open image file */
  if ( ( fp = fopen ( filename, "rb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file %s\n", filename );
    exit ( 1 );
  }

  /* read image */
  if ( read_TIFF ( fp, read_image ) ) {
    fprintf ( stderr, "error reading file %s\n", filename );
    exit ( 1 );
  }

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

  /* check the type of image data */
  if ( read_image->TIFF_type != type_check ) {
    fprintf ( stderr, "WARNING:  Wrong type: %s\n", read_image->TIFF_type);
  }
}

void write_routine(FILE *fp, char* filename, struct TIFF_img* write_image) {
    /* open color image file */
  if ( ( fp = fopen (filename, "wb" ) ) == NULL ) {
      fprintf ( stderr, "cannot open file %sf\n", filename);
      exit ( 1 );
  }

  /* write color image */
  if ( write_TIFF ( fp, write_image ) ) {
      fprintf ( stderr, "error writing TIFF file %s\n", filename );
      exit ( 1 );
  }

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

```c
void populate_tiff_from_img3(image3_t* img, struct TIFF_img* color_img) {
  int32_t i,j,pixel;

  for ( i = 0; i < img->height; i++ )
  for ( j = 0; j < img->width; j++ )
  for (int ch = 0; ch < 3; ch++) {
    pixel = (int32_t)img->img[ch][i][j];
    if(pixel>255) {
      color_img->color[ch][i][j] = 255;
    }
    else {
      if(pixel<0) color_img->color[ch][i][j] = 0;
      else color_img->color[ch][i][j] = pixel;
    }
  }
}

void allocate_img3(image3_t* target_img, int height, int width) {
  target_img->height =  height;
  target_img->width =  width;
  for (int ch = 0; ch < 3; ch++) {
    target_img->img[ch] = (double **)get_img(width,height,sizeof(double));
  }
}

int main (int argc, char **argv)
{
  FILE *fp = 0;
  struct TIFF_img input_img_tiff, filter_tiff, output_img_tiff;
  image3_t input_img, filtered_image, filter, filter_generated;
  int32_t i,j, part;
  float lambda;

  // Parse args:
  part = atoi(argv[1]);
  if ((argc == 4 && part == 2) ||
      (argc == 5 && part == 1)) {
     error(argv[0]);
  }
  open_routine(fp, argv[2], &input_img_tiff, 'c');

  /* Allocate image of double precision floats */

  allocate_img3(&input_img, input_img_tiff.height, input_img_tiff.width);
```

```
  allocate_img3(&filtered_image, input_img_tiff.height, input_img_tiff.width);

  /* copy all components */
  for ( i = 0; i < input_img.height; i++ )
  for ( j = 0; j < input_img.width; j++ )
  for (int ch = 0; ch < 3; ch++) {
    input_img.img[ch][i][j] = input_img_tiff.color[ch][i][j];
  }

  if (part == 1) {
    allocate_img3(&filter, 9, 9);
    for ( i = 0; i < filter.height; i++ )
    for ( j = 0; j < filter.width; j++ )
    for (int ch = 0; ch < 3; ch++) {
      filter.img[ch][i][j] = 1.0/81.0;
    }
    apply_filter_3(input_img, filter, filtered_image);
  }

  if (part == 2) {
    lambda = atof(argv[4]);
    printf("Lambda:  %f \n", lambda);

    allocate_img3(&filter, 5, 5);

    int fcenter_y =  filter.height / 2;
    int fcenter_x =  filter.width / 2;
    for ( i = 0; i < filter.height; i++ )
    for ( j = 0; j < filter.width; j++ ) {
      int delta  = (int)(i==fcenter_y && j==fcenter_x);
      for (int ch = 0; ch < 3; ch++) {
        filter.img[ch][i][j] = delta + lambda * (delta - 1.0/25.0);
      }
    }
    apply_filter_3(input_img, filter, filtered_image);
  }

  if (part == 3) {
    // NOTE:::
    // In tiff.c certain flag checks were removed (XResolution and YResolution) Line 2366 -
2367
    // Slow version (not runnng)
    if (0) {
      open_routine(fp, argv[4], &filter_tiff, 'g');
      allocate_img3(&filter_generated, filter_tiff.height, filter_tiff.width);
```

```
      for ( i = 0; i < filter_generated.height; i++ )
      for ( j = 0; j < filter_generated.width; j++ )
      for (int ch = 0; ch < 3; ch++) {
        filter_generated.img[ch][i][j] = filter_tiff.mono[i][j];
      }

      apply_filter_3(input_img, filter_generated, filtered_image);
    } else {
      // Fast version, just plainly apply the recursive relation:
      for ( i = 0; i < filtered_image.height; i++ )
      for ( j = 0; j < filtered_image.width; j++ )
      for (int ch = 0; ch < 3; ch++) {
        filtered_image.img[ch][i][j] = 0.01*input_img.img[ch][i][j];
        if (i) filtered_image.img[ch][i][j] += 0.9*filtered_image.img[ch][i-1][j];
        if (j) filtered_image.img[ch][i][j] += 0.9*filtered_image.img[ch][i][j-1];
        if (i&&j) filtered_image.img[ch][i][j] -= 0.81*filtered_image.img[ch][i-1][j-1];
      }
    }
  }

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

  //Save the image
  populate_tiff_from_img3(&filtered_image, &output_img_tiff);
  write_routine(fp, argv[3], &output_img_tiff);

  free_TIFF ( &(input_img_tiff) );
  free_TIFF ( &(output_img_tiff) );
  free_TIFF ( &(filter_tiff) );

  free_image_3(input_img);
  free_image_3(filtered_image);
  free_image_3(filter);


  printf("Success, exiting...\n");
  return(0);
}

void error(char *name)
{
    printf("usage: %s  part_no(1, 2, 3) input.tiff output.tiff [lambda_value] |
[filter_input]\n\n",name);
    printf("this program reads in a 24-bit color TIFF image.\n");
```

```
    printf("and a customfilter image.\n");
    printf("It then performs all tasks required in the lab\n");
    exit(1);
}
```