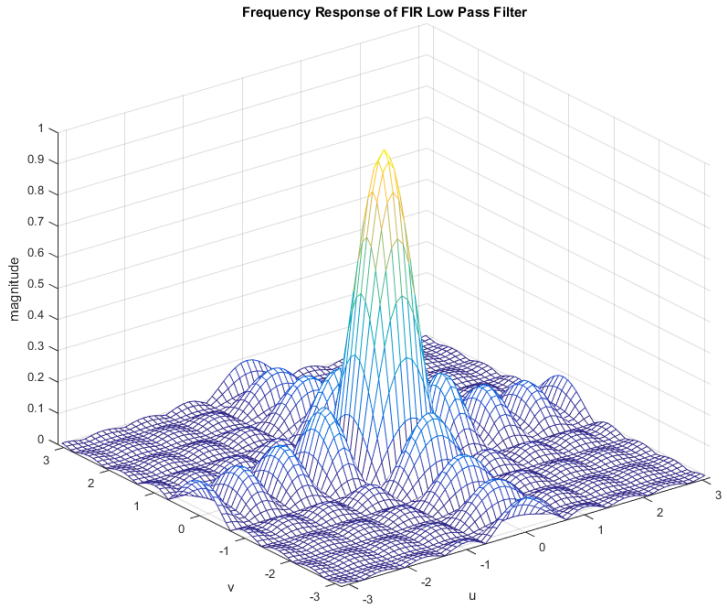
Lab 1 Report

Section 3 FIR Low Pass Filter

1. A derivation of the analytical expression for $H(e^{j\mu}, e^{jv})$.

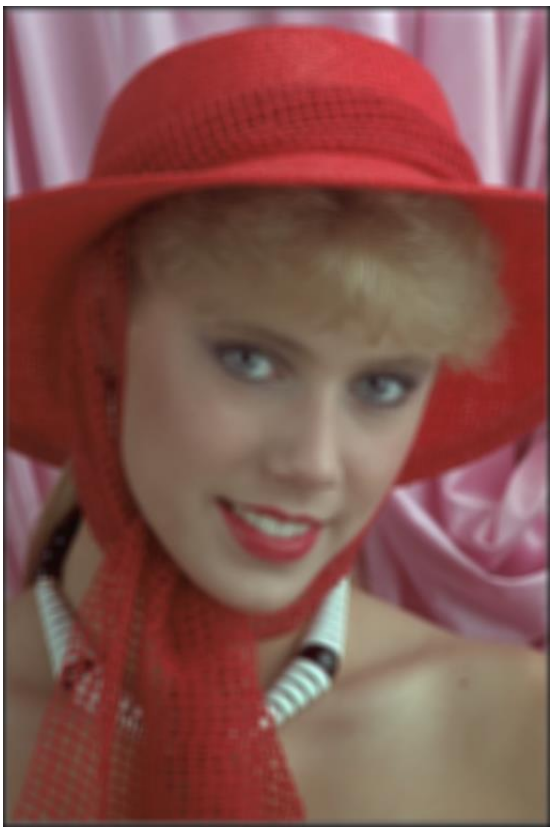
$$H(e^{j\mu}, e^{jv}) = \sum_{k=-4}^{4} \sum_{l=-4}^{4} \frac{1}{81} e^{-j(k\mu+lv)}$$

2. A plot of $|H(e^{j\mu}, e^{jv})|$.



3. The color image in imag03.tif

4. The filtered color image



5. A listing of C code

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

void fir_lowpass_filter(uint8_t **img, double **output, double **kernel, int
i, int j, int width, int height, int kernel_size);

void apply_color(struct TIFF_img output, double **input, int channel);

int main (int argc, char **argv)
{
  FILE *fp;
  struct TIFF_img input_img, filter_img;
  double **output;
  int kernel_size = 9;
  double **kernel;
  int32_t i, j;

  // check for argument count
  if ( argc != 2 ) {
    fprintf( stderr, "Missing Argument\n");
    exit(1);
  }
```

```
  //check for error in reading files
  if ( ( fp = fopen ( argv[1], "rb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file %s\n", argv[1] );
    exit ( 1 );
  }

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

  fclose(fp);

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

  //allocate memory
  output = (double **)get_img(input_img.width, input_img.height,
sizeof(double));
  kernel = (double **)get_img(kernel_size, kernel_size, sizeof(double));

  //create kernel
  printf("Create Kernel\n");
  for (i = 0; i < kernel_size; i++) {
    for (j = 0; j < kernel_size; j++) {
      kernel[i][j] = 1.0 / 81.0;
    }
  }

  //apply the filter
  printf("Apply filter\n");
  printf("Image size: %d %d\n", input_img.width, input_img.height);
  get_TIFF( &filter_img, input_img.height, input_img.width, 'c');
  for (int c = 0; c < 3; c++){
    for (i = 0; i < input_img.height; i++) {
      for (j = 0; j < input_img.width; j++) {
        fir_lowpass_filter(input_img.color[c], output, kernel, i, j,
                           input_img.width, input_img.height, kernel_size);
      }
    }
    printf("Channel %d complete\n", c);
    apply_color(filter_img, output, c);
    printf("Applied channel %d color\n", c);
  }

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

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

```
    exit ( 1 );
  }

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

  /* de-allocate memory */
  free_TIFF(&(input_img));
  free_TIFF(&(filter_img));
  free_img((void**)output);
  free_img((void**)kernel);
}

void fir_lowpass_filter(uint8_t **img, double **output, double **kernel, int
i, int j, int width, int height, int kernel_size)
{
  double sum = 0.0;
  for (int k = 0; k < kernel_size; k++) {
    for (int l = 0; l < kernel_size; l++) {
      int loc_i = i + k - kernel_size / 2;
      int loc_j = j + l - kernel_size / 2;
      if (loc_i >= 0 && loc_i < height && loc_j >= 0 && loc_j < width) {
        sum += kernel[k][l] * img[loc_i][loc_j];
      }
    }
  }
  output[i][j] = sum;
}

void apply_color(struct TIFF_img output, double **input, int channel)
{
  for (int i = 0; i < output.height; i++) {
    for (int j = 0; j < output.width; j++) {
      int32_t pixel = (int32_t)input[i][j];
      if (pixel > 255) {
        pixel = 255;
      }
      output.color[channel][i][j] = (int32_t)input[i][j];
    }
  }
}
```

Section 4 FIR Sharpening Filter

1. A derivation of the analytical expression for $H(e^{j\mu}, e^{jv})$.

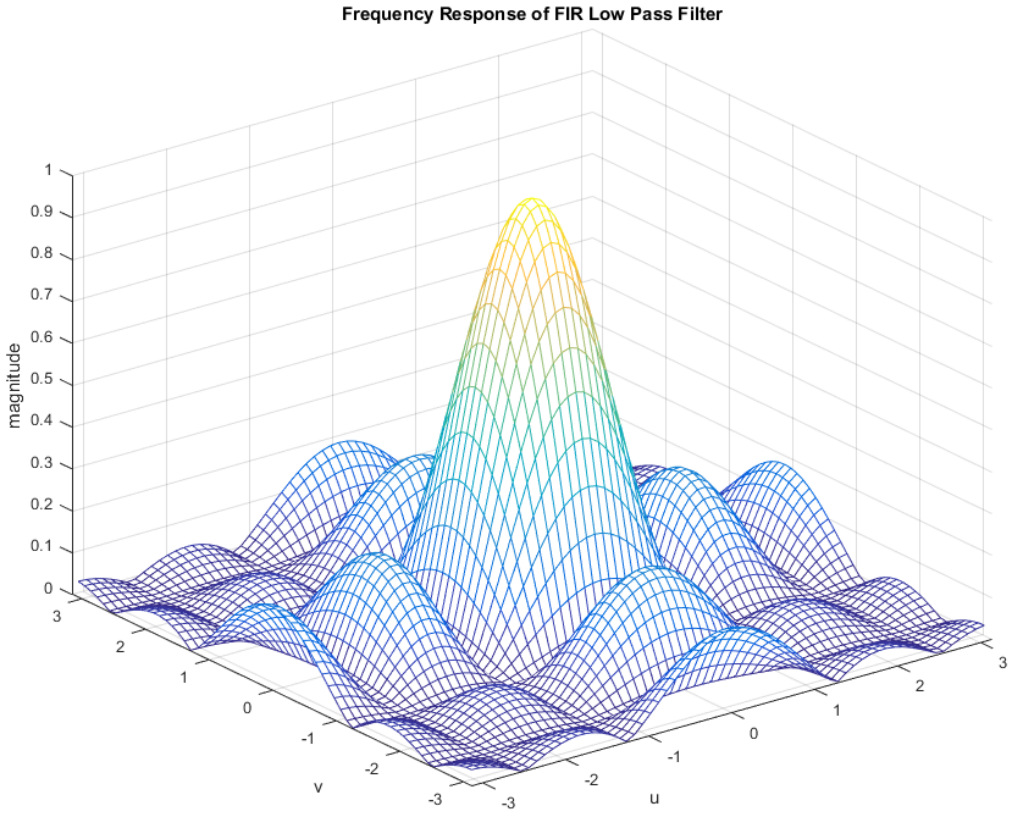
$$H(e^{j\mu}, e^{jv}) = \sum_{k=-2}^{2} \sum_{l=-2}^{2} \frac{1}{25} e^{-j(k\mu+lv)}$$

2. A derivation of the analytical expression for $G(e^{j\mu}, e^{jv})$.

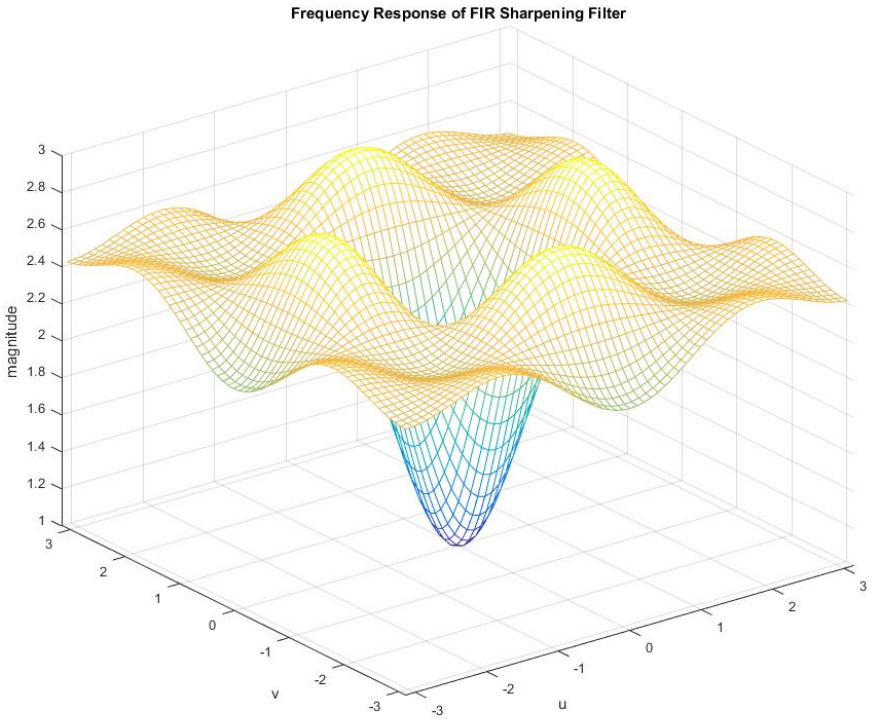
$$g(m,n) = \delta(m,n) + \lambda(\delta(m,n) - h(m,n))$$

$$G(e^{j\mu}, e^{jv}) = 1 + \lambda\left(1 - H(e^{j\mu}, e^{jv})\right) = 1 + \lambda\left(1 - \sum_{k=-2}^{2} \sum_{l=-2}^{2} \frac{1}{25} e^{-j(k\mu+lv)}\right)$$

3. A plot of $\left|H(e^{j\mu}, e^{jv})\right|$.



4. A plot of $\left|G\left(e^{j\mu}, e^{jv}\right)\right|$ for $\lambda = 1.5$



5. The input color image imgblur.tif.

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

7. A listing of C code.

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

void fir_filter(uint8_t **img, double **output, double **kernel, int i, int
j, int width, int height, int kernel_size);
void apply_color(struct TIFF_img output, double **input, int channel);

int main (int argc, char **argv)
{
  FILE *fp;
  struct TIFF_img input_img, filter_img;
  double **output;
  int kernel_size = 5;
  double **kernel;
  int32_t i, j;
  double lambda;
  // check for argument count
  if ( argc != 3 ) {
    fprintf( stderr, "Missing Argument\n");
```

```
    exit(1);
  }

  //check for error in reading files
  if ( ( fp = fopen ( argv[1], "rb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file %s\n", argv[1] );
    exit ( 1 );
  }

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

  fclose(fp);

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

  sscanf(argv[2], "%lf", &lambda);

  //allocate memory
  output = (double **)get_img(input_img.width, input_img.height,
sizeof(double));
  kernel = (double **)get_img(kernel_size, kernel_size, sizeof(double));

  //create kernel
  printf("Create Kernel\n");
  for (i = 0; i < kernel_size; i++) {
    for (j = 0; j < kernel_size; j++) {
      if (i == kernel_size / 2 && j == kernel_size / 2) {
        kernel[i][j] = 1.0 + lambda * (1.0 - 1.0/25.0);
      } else {
        kernel[i][j] = lambda * (-1.0/25.0);
      }
    }
  }

  for (i = 0; i < kernel_size; i++) {
    for (j = 0; j < kernel_size; j++) {
      printf("%f,", kernel[i][j]);
    }
    printf("\n");
  }
  //apply the filter
  printf("Apply filter\n");
  printf("Image size: %d %d\n", input_img.width, input_img.height);
  get_TIFF( &filter_img, input_img.height, input_img.width, 'c');
  for (int c = 0; c < 3; c++){
    for (i = 0; i < input_img.height; i++) {
      for (j = 0; j < input_img.width; j++) {
        fir_filter(input_img.color[c], output, kernel, i, j, input_img.width,
input_img.height, kernel_size);
      }
```

```
    }
    printf("Channel %d complete\n", c);
    apply_color(filter_img, output, c);
    printf("Applied channel %d color\n", c);
  }

  /* open image file for write */

  if ( ( fp = fopen ( "sharpen.tif", "wb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file lowpass_filter.tif\n");
    exit ( 1 );
  }

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

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

  /* de-allocate memory */
  free_TIFF(&(input_img));
  free_TIFF(&(filter_img));
  free_img((void**)output);
  free_img((void**)kernel);
}

void fir_filter(uint8_t **img, double **output, double **kernel, int i, int
j, int width, int height, int kernel_size)
{
  double sum = 0.0;
  for (int k = 0; k < kernel_size; k++) {
    for (int l = 0; l < kernel_size; l++) {
      int loc_i = i + k - kernel_size / 2;
      int loc_j = j + l - kernel_size / 2;
      if (loc_i >= 0 && loc_i < height && loc_j >= 0 && loc_j < width) {
        sum += kernel[k][l] * img[loc_i][loc_j];
      }
    }
  }
  output[i][j] = sum;
}

void apply_color(struct TIFF_img output, double **input, int channel)
{
  for (int i = 0; i < output.height; i++) {
    for (int j = 0; j < output.width; j++) {
      int32_t pixel = (int32_t)input[i][j];
      if (pixel > 255) {
        pixel = 255;
      }
      output.color[channel][i][j] = (int32_t)input[i][j];
    }
  }
}
```

Section 5 IIR Filter

1. A derivation of the analytical expression for $H(e^{j\mu}, e^{jv})$.

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

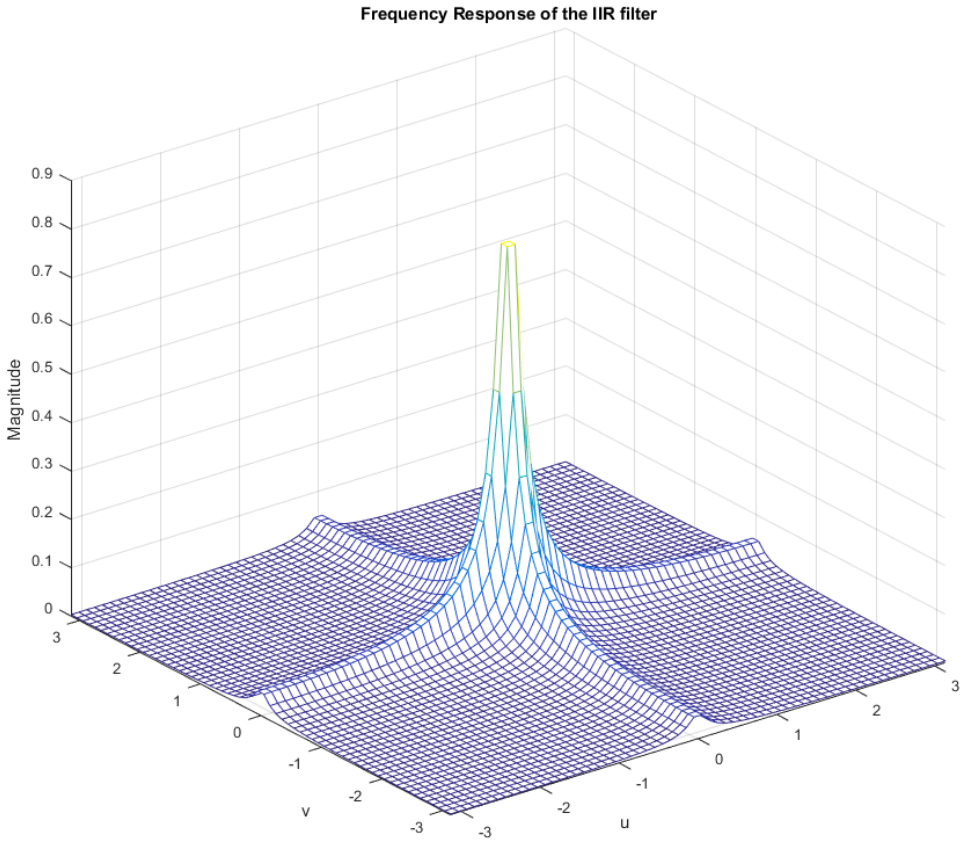
$$Y(z_1,z_2) = 0.01X(m,n) + 0.9\big(z_1^{-1}Y(z_1,z_2) + z_2^{-1}Y(z_1,z_2)\big) - 0.81z_1^{-1}z_2^{-2}Y(z_1,z_2)$$

$$(1 - 0.9z_1^{-1} - 0.9z_2^{-1} + 0.81z_1^{-1}z_2^{-1})Y(z_1,z_2) = 0.01X(z_1,z_2)$$

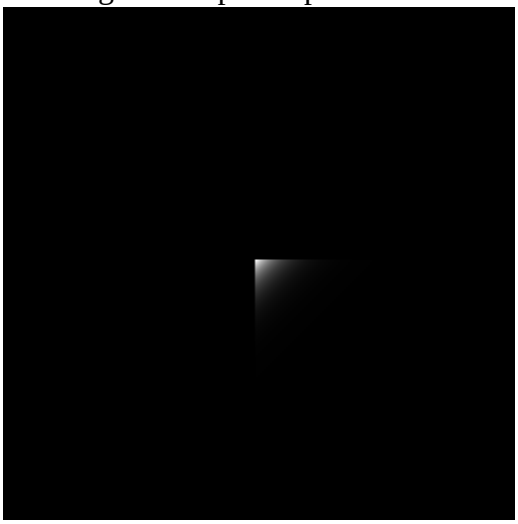
$$H(z_1,z_2) = \frac{Y(z_1,z_2)}{X(z_1,z_2)} = \frac{0.01}{1 - 0.9z_1^{-1} - 0.9z_2^{-1} + 0.81z_1^{-1}z_2^{-1}}$$

$$H\big(e^{ju}, e^{jv}\big) = \frac{0.01}{1 - 0.9e^{-ju} - 0.9e^{-jv} + 0.81e^{-(ju+jv)}}$$

2. A plot of $\big|H\big(e^{j\mu}, e^{jv}\big)\big|$.



3. An image of the point spread function.



4. The filtered output color image

5. A listing of C code.

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

void iir_filter(uint8_t **img, double **output, int i, int j, int width, int
height);

void apply_color(struct TIFF_img output, double **input, int channel);

int main (int argc, char **argv)
{
  FILE *fp;
  struct TIFF_img input_img, filter_img, psf_img;
  double **output;
  double **kernel;
  int32_t i, j;

  // check for argument count
  if ( argc != 3 ) {
    fprintf( stderr, "Missing Argument\n");
    exit(1);
  }

  //check for error in reading files
  if ( ( fp = fopen ( argv[1], "rb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file %s\n", argv[1] );
    exit ( 1 );
  }

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

  fclose(fp);

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

  if ((fp = fopen(argv[2], "rb")) == NULL) {
    fprintf ( stderr, "cannot open file %s\n", argv[2] );
    exit ( 1 );
  }

  if (read_TIFF(fp, &psf_img)) {
    fprintf( stderr, "error reading file %s\n", argv[2] );
    exit(1);
  }

  fclose(fp);
```

```
  //allocate memory
  output = (double **)get_img(input_img.width, input_img.height,
sizeof(double));
  kernel = (double **)get_img(psf_img.width, psf_img.height, sizeof(double));

  //covert tif to kernel
  printf("Create Kernel\n");
  for (i = 0; i < psf_img.width; i++) {
    for (j = 0; j < psf_img.width; j++) {
      kernel[i][j] = psf_img.mono[i][j] / 255.0 / 100.0;
    }
  }

  //apply the filter
  printf("Apply filter\n");
  printf("Image size: %d %d\n", input_img.width, input_img.height);
  get_TIFF( &filter_img, input_img.height, input_img.width, 'c');
  for (int c = 0; c < 3; c++){
    for (i = 0; i < input_img.height; i++) {
      for (j = 0; j < input_img.width; j++) {
        iir_filter(input_img.color[c], output, i, j, input_img.width,
input_img.height);
      }
    }
    printf("Channel %d complete\n", c);
    apply_color(filter_img, output, c);
    printf("Applied channel %d color\n", c);
  }

  /* open image file for write */

  if ( ( fp = fopen ( "iir_filter.tif", "wb" ) ) == NULL ) {
    fprintf ( stderr, "cannot open file lowpass_filter.tif\n");
    exit ( 1 );
  }

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

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

  /* de-allocate memory */
  free_TIFF(&(input_img));
  free_TIFF(&(filter_img));
  free_TIFF(&(psf_img));
  free_img((void**)output);
  free_img((void**)kernel);
}

void iir_filter(uint8_t **img, double **output, int i, int j, int width, int
height)
{
```

```
  double sum = 0.0;
  sum = 0.01 * img[i][j];
  if (i - 1 >= 0) {
    sum += 0.9 * output[i-1][j];
  }
  if (j - 1 >= 0) {
    sum += 0.9 * output[i][j-1];
  }
  if (i - 1 >= 0 && j - 1 >= 0) {
    sum -= 0.81 * output[i-1][j-1];
  }
  output[i][j] = sum;
}

void apply_color(struct TIFF_img output, double **input, int channel)
{
  for (int i = 0; i < output.height; i++) {
    for (int j = 0; j < output.width; j++) {
      int32_t pixel = (int32_t)input[i][j];
      if (pixel > 255) {
        pixel = 255;
      }
      output.color[channel][i][j] = (int32_t)input[i][j];
    }
  }
}
```