Image Processing Laboratory

Experiment 4: Frequency Filtering

Report

Group-6

Submitted by

Deep Kiran Shroti (12EC35013)
Vishnu Dutt Sharma (12EC35018)

Contents

1 Introduction

The objective of this experiment is to apply following *Frequency Filtering* operations on an image:

1. Ideal Low-pass Filter
2. Ideal High-pass Filter
3. Butterworth Low-pass Filter
4. Butterworth High-pass Filter
5. Gaussian Low-pass Filter
6. Gaussian High-pass Filter

Each of the above filter helps in modification or extracting features by maipulating frequency spectrum of the image. The image is Fourier transformed, multiplied with the filter function and then re-transformed into the spatial domain. Supressing low frequencies result in a enhancement of edges, while supressing high frequencies results in a smoother image in the spatial domain.
All frequency filters can also be implemented in the spatial domain and, if there exists a simple kernel for the desired filter effect, it is computationally less expensive to perform the filtering in the spatial domain. Frequency filtering is more appropriate if no straightforward kernel can be found in the spatial domain, and may also be more efficient.
Functions for various filters are as follows:

- *Ideal Low-pass Filter*: In the following function, C_x and C_y are the cutoff frequencies for the frequency spectrum in x-direction and y-direction respectively.

$$H(\omega_x, \omega_y) = \begin{cases} 1 & \text{if } \omega_x \le C_x, \omega_y \le C_y \\ 0 & otherwise \end{cases}$$

- *Ideal High-pass Filter*: In the following function, C_x and C_y are the cutoff frequencies for the frequency spectrum in x-direction and y-direction respectively.

$$H(\omega_x, \omega_y) = \begin{cases} 1 & \text{if } \omega_x \ge C_x, \omega_y \ge C_y \\ 0 & otherwise \end{cases}$$

- *Butterworth Low-pass Filter*: In the following function, C and n are the parameters that decide the spred and steepness of the function

$$H(\omega_x, \omega_y) = \frac{1}{1 + \left(\frac{\omega_x^2 + \omega_y^2}{c_2}\right)^{2n}}$$

- *Butterworth High-pass Filter*: In the following function, C and n are the parameters that decide the spred and steepness of the function

$$H(\omega_x, \omega_y) = 1 - \frac{1}{1 + \left(\frac{\omega_x^2 + \omega_y^2}{c_2}\right)^{2n}}$$

- *Gaussian Low-pass Filter*: In the following function, σ is the square root of variance ($varience = \sigma^2$)

$$H(\omega_x, \omega_y) = e^{-\frac{\omega_x^2 + \omega_y^2}{2\sigma^2}}$$

- *Gaussian High-pass Filter*: In the following function, σ is the square root of variance ($varience = \sigma^2$)

$$H(\omega_x, \omega_y) = 1 - e^{-\frac{\omega_x^2 + \omega_y^2}{2\sigma^2}}$$

2 Algorithm

2-D FFT requires 1-D FFT on rows first and then on columns. To do this, we first perform FFT on each row using Divide and Conquer sttrategy. The we transpose this coefficiemt matrix, and perfom FFT on rows again. Same strategy is used for Inverse FFT.
We calculate filter coefficients by using the functions as mentioned above.
Code for 2-D FFT is as follows:

```
complex<double> **FFT2( string filename ){
  // temporary variable
  complex<float> temp;
  // Opening image
  Mat img = imread(filename, IMREAD_GRAYSCALE);

  // Variable for storing input and putput
  complex<double> **inpData = new complex<double>* [ img.rows ];
  complex<double> **fftData = new complex<double>* [ img.rows ];

  if (!img.data){
    cout << "Error:Image␣not␣found" <<endl;
    return inpData;
  }

  // Initializaing arrays
  for(int i = 0; i < img.rows; i++){
    inpData[i] = new complex<double> [ img.cols ];
    fftData[i] = new complex<double> [ img.cols ];
  }
  // Loading image data
  for(int i = 0; i < img.rows; i++)
    for(int j = 0; j < img.cols; j++)
      inpData[i][j] = complex<double>( (double)img.at<uchar>(i,j), 0.0);


  // Performing FFT on each row
  for( int i = 0; i < img.rows; i++)
    FFT( img.cols, inpData[i], fftData[i]);

  // Transpose the image
  for(int i = 0; i < img.rows; i++){
    for( int j = i; j < img.cols; j++){
      temp = fftData[i][j];
      fftData[i][j] = fftData[j][i];
      fftData[j][i] = temp;
    }
  }

  // Perform FFt on each row (column previously)
  for(int i = 0; i < img.rows; i++)
    FFT(img.cols, fftData[i], inpData[i]);

  // Deallocate memory
  for(int i = 0; i < img.rows; i++)
    delete[] fftData[i];
  delete[] fftData;

  // Return the transformed data
  return inpData;
}
```

Code for 2-D Inverse FFT is as follows:

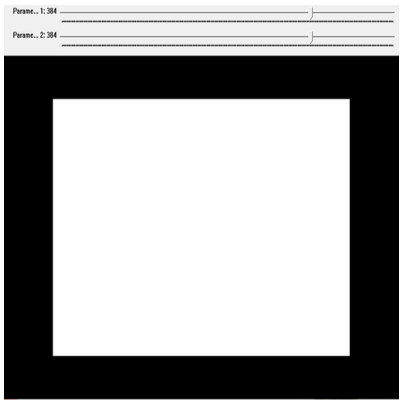
```
void IFFT2( complex<double> *inpData[COL_SIZE] ){
  // Temporary variable
  complex<float> temp;

  // Create a grayscale image
  Mat img(ROW_SIZE, COL_SIZE, CV_8UC1);

  complex<double> **fftData = new complex<double>* [ img.rows ];

  // Loading input data
  for(int i = 0; i < img.rows; i++){
    fftData[i] = new complex<double> [ img.cols ];
    }

    // Perform Inverse FFT on each row
    for( int i = 0; i < img.rows; i++)
      IFFT( img.cols, inpData[i], fftData[i]);

    // Transpose the data
    for(int i = 0; i < img.rows; i++){
      for( int j = i; j < img.cols; j++){
        temp = fftData[i][j];
        fftData[i][j] = fftData[j][i];
        fftData[j][i] = temp;
      }
    }

    // Perform FFT on each row (column previously)
    for(int i = 0; i < img.rows; i++)
      IFFT(img.cols, fftData[i], inpData[i]);

    // Load the data into image
    for(int i = 0; i < img.rows; i++)
      for(int j = 0; j < img.cols; j++)
        img.at<uchar>(i,j) = (unsigned char)((inpData[i][j]).real());


    // Deallocate memory
    for(int i = 0; i < img.rows; i++)
      delete[] fftData[i];
    delete[] fftData;

    // Show the image
    imshow("Output␣Image", img);
  }
```

FFT and IFFT are calculated in similar ways with change in power of omega only. Thus, we're giving only the FFT code here

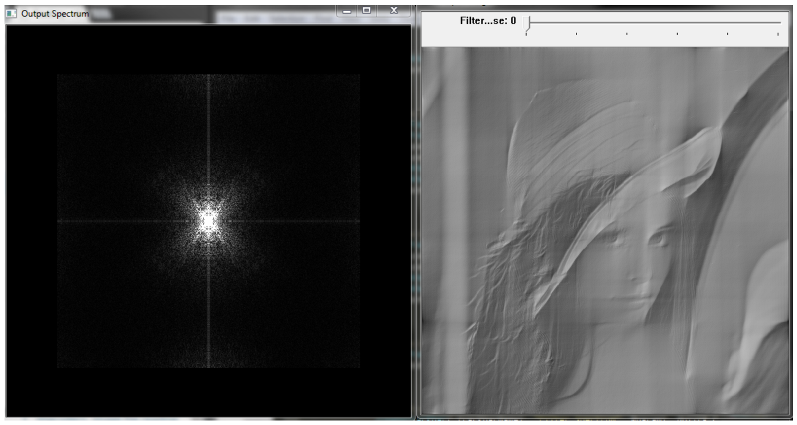
```
  void FFT(int n, complex<double> InArray[], complex<double> OutArray[]){
    // Variables
    int j,k, shft = n/2;
    complex<double> intermed;

    // Base case, only one element
    if(n == 1)
    {
      OutArray[0] = InArray[0];
      return;
  }

  // Arrays for storing parts of FFT
  complex<double> *E, *O, *EF, *OF;
  // Even terms
  E = ( complex<double> *)malloc( (n/2) * sizeof( complex<double>) );
  // Odd terms
  O = ( complex<double> *)malloc( (n/2) * sizeof( complex<double>) );
  // FFT coefficients of even terms
  EF = ( complex<double> *)malloc( (n/2) * sizeof( complex<double>) );
  // FFT coefficients of odd terms
  OF = ( complex<double> *)malloc( (n/2) * sizeof( complex<double>) );

  // Storing Even and Odd terms
  for( j = k = 0; k < n; j++, k += 2){
    E[j] = InArray[k];
    O[j] = InArray[k+1];
  }

  // Perform FFT on even and odd terms recursively
  FFT( n/2, E, EF);
  FFT( n/2, O, OF);

  complex<double> Omega(1.0, 0.0);
  complex<double> omega_base(cos(2*M_PI/n), -sin(2*M_PI/n));


  // Csalculation using even and odd terms
  for( k = 0; k < n/2; k++, Omega = Omega * omega_base)
  {
    intermed = Omega * OF[k];

    OutArray[k ] = EF[k] + intermed;

    OutArray[k + shft] = EF[k] - intermed;


  }

  // Deallocate memory
  delete[] E;
  delete[] O;
  delete[] EF;
  delete[] OF;
}
```

3 Output Results

Following are the results for filters mentioned above:

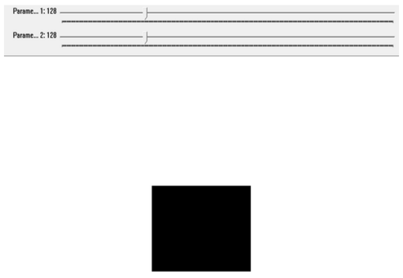


(a) Frequency spectrum of filter

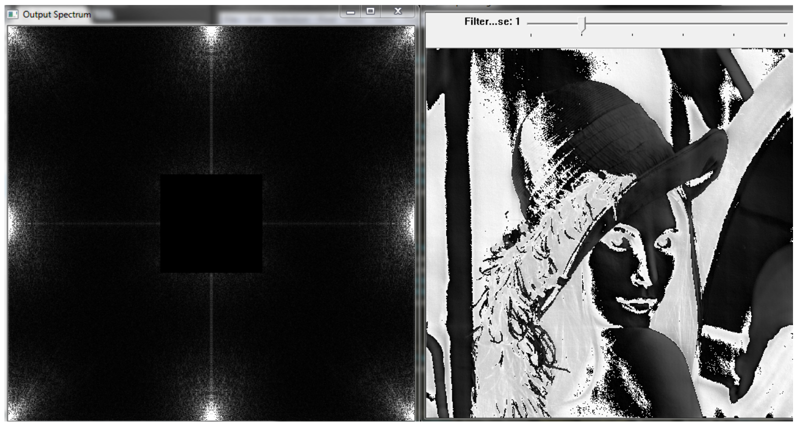


(b) Output Response and Image

Figure 1: Ideal Low-pass Filter

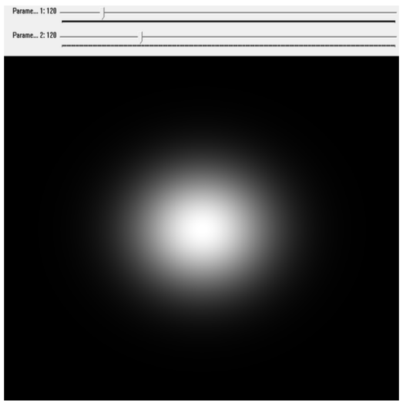


(a) Frequency spectrum of filter

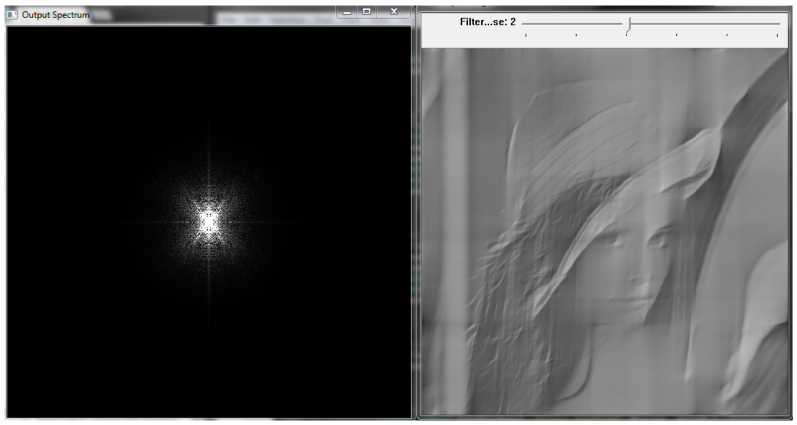


(b) Output Response and Image

Figure 2: Ideal High-pass Filter

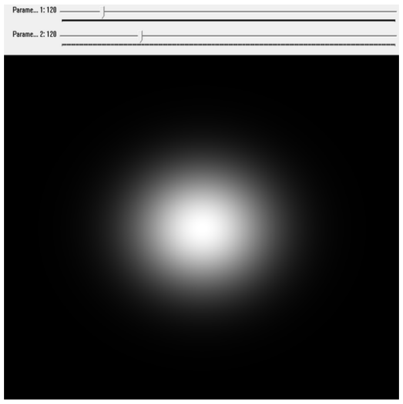


(a) Frequency spectrum of filter



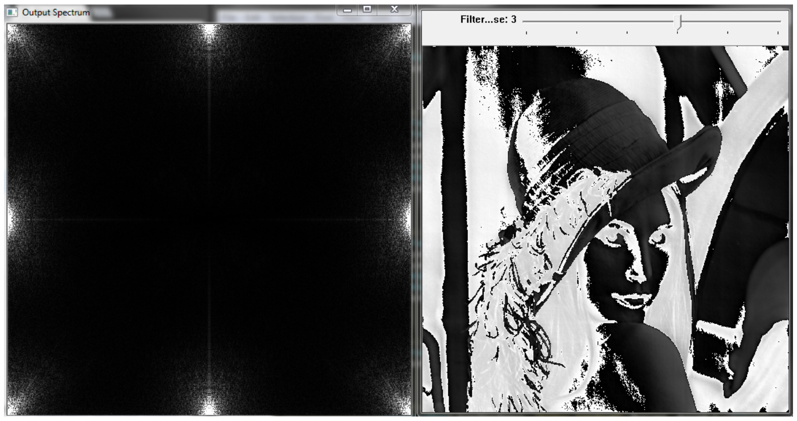
(b) Output Response and Image

Figure 3: Gaussian Low-pass Filter


Parame... 1: 120

Parame... 2: 120

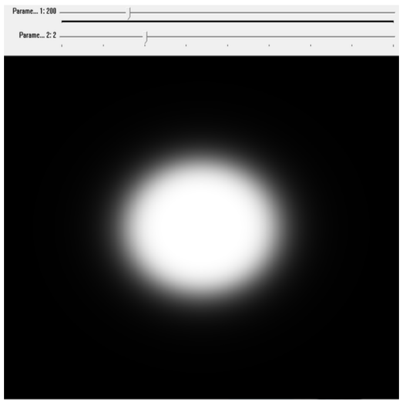
(a) Frequency spectrum of filter



Filter...se: 3

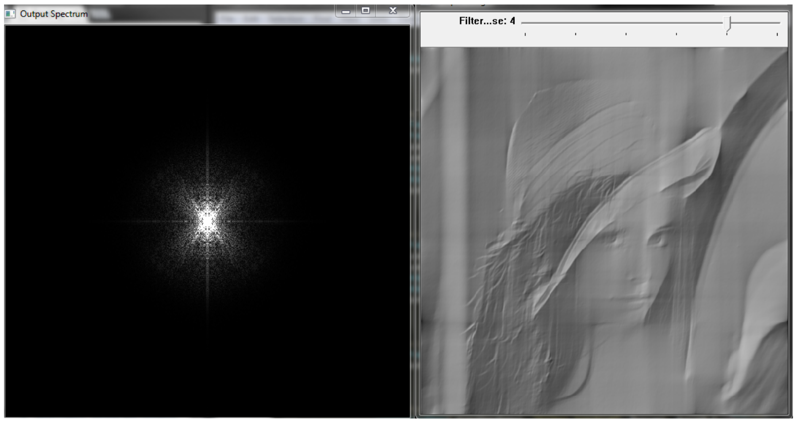
(b) Output Response and Image

Figure 4: Gaussian High-pass Filter


Parame... 1: 200

Parame... 2: 2

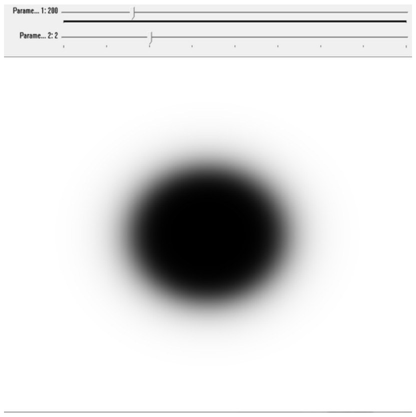
(a) Frequency spectrum of filter



Filter...se: 4

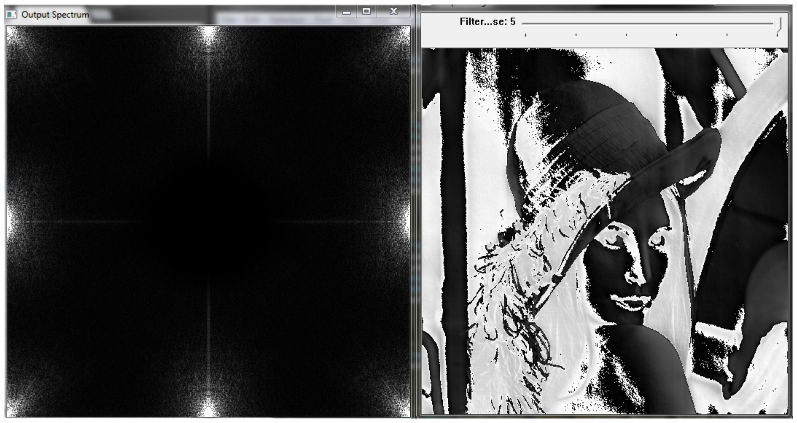
(b) Output Response and Image

Figure 5: Butterworth Low-pass Filter


Parame... 1: 200

Parame... 2: 2

(a) Frequency spectrum of filter



Filter...se: 5

(b) Output Response and Image

Figure 6: Butterworth High-pass Filter

Note: Parameters can be observed on the slider.

4 Analysis

- Frequency domain filtering provides more control over frequency. For example, in an ideal low-pass filter we can remove high frequency components easily. Thus they are easier for filtering operations.

- Frequency filtering are more computationally expensive than the spatial filters because of the requirement of fourier transform and inverse fourier transform. Thus they are use only in case we can't create a mask in spatial domain for an operation.

- Ideal filters suffer from ringing (Gibbs) phenomenon. It means some similar to oscillation response are observed in the output. It is due to non-continues nature of the filter.

- To avoid ringing effects in ideal filters which occurs due to discontinuity in filter response, Butterworth and Gaussian filters are used which are continuous. The parameters(like σ or order) decide how close they are to the ideal filters by affecting the slope in transition band.

5 Sources

[1] Course notes on Image Processing and Reconstruction by UCSC
`https://classes.soe.ucsc.edu/ee264/Fall11/LecturePDF/8-SpectralFiltering.pdf`

[2] Wikipedia page on Butterworth filter
`https://en.wikipedia.org/wiki/Butterworth _ filter`

[3] Wikipedia page on Gaussian Filter
`https://en.wikipedia.org/wiki/Gaussian_ filter`