Image Processing Lab

Batch 9:

Ajay Viswanathan [09EC3509]

Sourin Sutradhar [09EC3514]

Experiment 6

Frequency Filtering

Problem Objective:

Write C++/Image-J modular functions to perform the following operations on the 512 × 512 grayscale Lena image.

1. FFT2 (takes input image filename as the argument, and gives 2D FFT coefficients as output)

2. IFFT2 (takes 2D FFT coefficients as input argument, and gives the back-projected / reconstructed image as output)

3. Perform Ideal, Gaussian, and Butterworth low-pass and high-pass filtering, taking cut-off frequency, D0, and image filename as input arguments) respectively with

- Ideal_LPF Ideal_HPF
- Gaussian_LPF Gaussian_HPF
- Butterworth_LPF Butterworth_HPF

Document your observations on various types of filtering.

Brief Theory:

A fast Fourier transform (FFT) is an algorithm to compute the discrete Fourier transform (DFT) and its inverse. There are many different FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory; this article gives an overview of the available techniques and some of their general properties, while the specific algorithms are described in subsidiary articles linked below.

The DFT is obtained by decomposing a sequence of values into components of different frequencies. This operation is useful in many fields (see discrete Fourier transform for properties and applications of the transform) but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly: computing the DFT of N points in the naive way, using the definition, takes O(N2) arithmetical operations, while an FFT can compute the same DFT in only O(N log N) operations. The difference in speed can be enormous, especially for long data sets where N may be in the thousands or millions. In practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to N / log(N). This huge improvement made the calculation of the DFT practical; FFTs are of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers.

The best-known FFT algorithms depend upon the factorization of N, but there are FFTs with O(N log N) complexity for all N, even for prime N. Many FFT algorithms only depend on the fact that is an N-th primitive root of unity, and thus can be applied to analogous transforms over any finite field, such as number-theoretic transforms. Since the inverse DFT is the same as the DFT, but with the opposite sign in the exponent and a 1/N factor, any FFT algorithm can easily be adapted for it.

An FFT computes the DFT and produces exactly the same result as evaluating the DFT definition directly; the only difference is that an FFT is much faster. (In the presence of round-off error, many FFT algorithms are also much more accurate than evaluating the DFT definition directly, as discussed below.)

Let x0, ..., xN-1 be complex numbers. The DFT is defined by the formula

Evaluating this definition directly requires O(N2) operations: there are N outputs Xk, and each output requires a sum of N terms. An FFT is any method to compute the same results in O(N log N) operations. More precisely, all known FFT algorithms require Θ(N log N) operations (technically, O only denotes an upper bound), although there is no known proof that a lower complexity score is impossible.

To illustrate the savings of an FFT, consider the count of complex multiplications and additions. Evaluating the DFT's sums directly involves N2 complex multiplications and N(N−1) complex additions [of which O(N) operations can be saved by eliminating trivial operations such as multiplications by 1]. The well-known radix-2 Cooley–Tukey algorithm, for N a power of 2, can compute the same result with only (N/2)log2(N) complex multiplies (again, ignoring simplifications of multiplications by 1 and similar) and Nlog2(N) complex additions. In practice, actual performance on modern computers is usually dominated by factors other than the speed of arithmetic operations and the analysis is a complicated subject (see, e.g., Frigo & Johnson, 2005), but the overall improvement from O(N2) to O(N log N) remains.

The FFT operates by decomposing an N point time domain signal into N time domain signals each composed of a single point. The second step is to calculate the N frequency spectra corresponding to these N time domain signals. Lastly, the N spectra are synthesized into a single frequency spectrum.

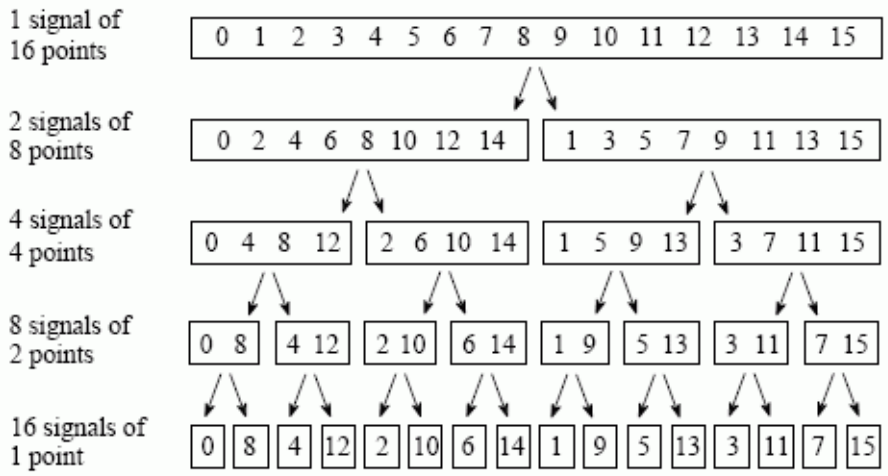


FIGURE 12-2
The FFT decomposition. An *N* point signal is decomposed into *N* signals each containing a single point. Each stage uses an *interlace decomposition*, separating the even and odd numbered samples.

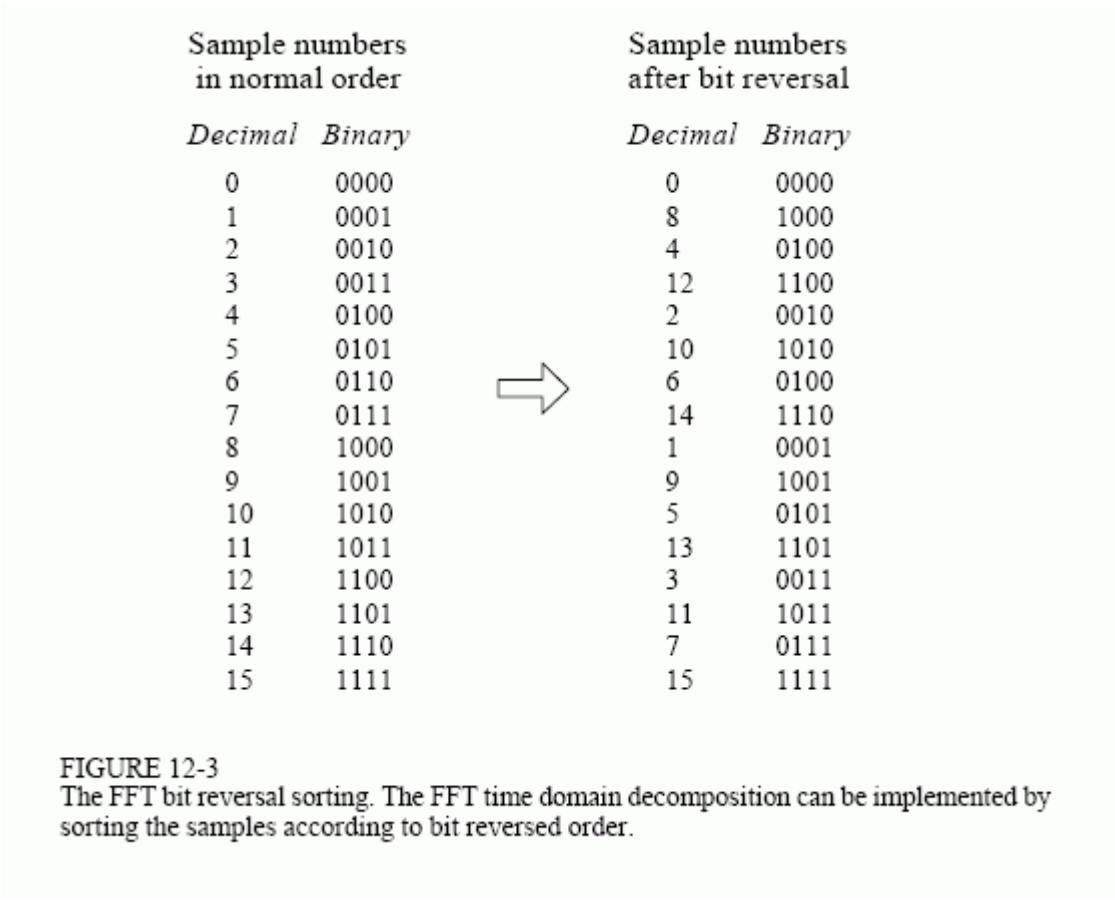


FIGURE 12-3
The FFT bit reversal sorting. The FFT time domain decomposition can be implemented by sorting the samples according to bit reversed order.

In this example, a 16 point signal is decomposed through four separate stages. The first stage breaks the 16 point signal into two signals each consisting of 8 points. The second stage decomposes the data into four signals of 4 points. This pattern continues until there are N signals composed of a single point. An interlaced decomposition is used each time a signal is broken in two, that is, the signal is separated into its even and odd numbered samples. The best way to understand this is by inspecting Fig. 12-2 until you grasp the pattern. There are Log2N stages required in this decomposition, i.e., a 16 point signal (24) requires 4 stages, a 512 point signal (27) requires 7 stages, a 4096 point signal (212) requires 12 stages, etc. Remember this value, Log2N; it will be referenced many times in this chapter.

Now that you understand the structure of the decomposition, it can be greatly simplified. The decomposition is nothing more than a reordering of the samples in the signal. Figure 12-3 shows the rearrangement pattern required. On the left, the sample numbers of the original signal are listed along with their binary equivalents. On the right, the rearranged sample numbers are listed, also along with their binary equivalents. The important idea is that the binary numbers are the reversals of each other. For example, sample 3 (0011) is exchanged with sample number 12 (1100). Likewise, sample number 14 (1110) is swapped with sample number 7 (0111), and so forth. The FFT time domain decomposition is usually carried out by a bit reversal sorting algorithm. This involves rearranging the order of the N time domain samples by counting in binary with the bits flipped left-for-right (such as in the far right column in Fig. 12-3).

The next step in the FFT algorithm is to find the frequency spectra of the 1 point time domain signals. Nothing could be easier; the frequency spectrum of a 1 point signal is equal to itself. This means that nothing is required to do this step. Although there is no work involved, don't forget that each of the 1 point signals is now a frequency spectrum, and not a time domain signal.

The last step in the FFT is to combine the N frequency spectra in the exact reverse order that the time domain decomposition took place. This is where the algorithm gets messy. Unfortunately, the bit reversal shortcut is not applicable, and we must go back one stage at a time. In the first stage, 16 frequency spectra (1 point each) are synthesized into 8 frequency spectra (2 points each). In the second stage, the 8 frequency spectra (2 points each) are synthesized into 4 frequency spectra (4 points each), and so on. The last stage results in the output of the FFT, a 16 point frequency spectrum.

Figure 12-4 shows how two frequency spectra, each composed of 4 points, are combined into a single frequency spectrum of 8 points. This synthesis must undo the interlaced decomposition done in the time domain. In other words, the frequency domain operation must correspond to the time domain procedure of combining two 4 point signals by interlacing.

Consider two time domain signals, abcd and efgh. An 8 point time domain signal can be formed by two steps: dilute each 4 point signal with zeros to make it an 8 point signal, and then add the signals together. That is, abcd becomes a0b0c0d0, and efgh becomes 0e0f0g0h. Adding these two 8 point signals produces aebfcgdh. As shown in Fig. 12-4, diluting the time domain with zeros corresponds to a duplication of the frequency spectrum. Therefore, the frequency spectra are combined in the FFT by duplicating them, and then adding the duplicated spectra together. Figure 12-5 shows a flow diagram for combining two 4 point spectra into a single 8 point spectrum. To reduce the situation even more, notice that Fig. 12-5 is formed from the basic pattern in Fig 12-6 repeated over and over.

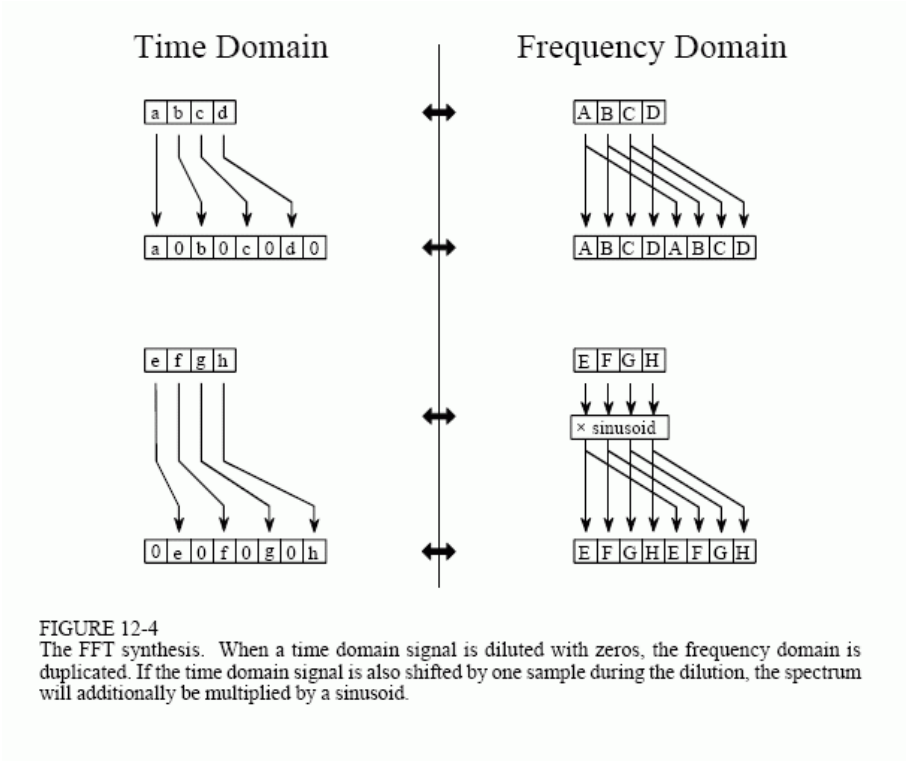


FIGURE 12-4
The FFT synthesis. When a time domain signal is diluted with zeros, the frequency domain is duplicated. If the time domain signal is also shifted by one sample during the dilution, the spectrum will additionally be multiplied by a sinusoid.

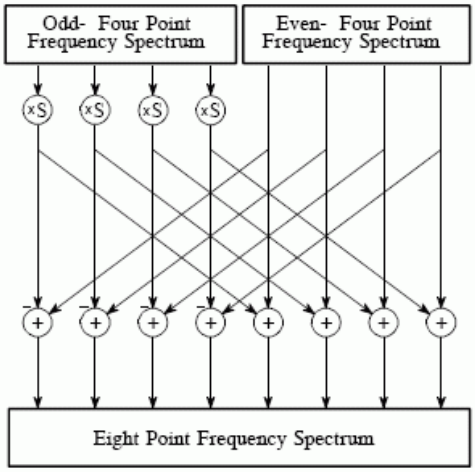


FIGURE 12-5
FFT synthesis flow diagram. This shows the method of combining two 4 point frequency spectra into a single 8 point frequency spectrum. The ×S operation means that the signal is multiplied by a sinusoid with an appropriately selected frequency.

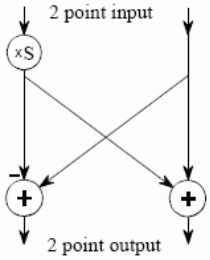


FIGURE 12-6
The FFT butterfly. This is the basic calculation element in the FFT, taking two complex points and converting them into two other complex points.

This simple flow diagram is called a butterfly due to its winged appearance. The butterfly is the basic computational element of the FFT, transforming two complex points into two other complex points.

Figure 12-7 shows the structure of the entire FFT. The time domain decomposition is accomplished with a bit reversal sorting algorithm. Transforming the decomposed data into the frequency domain involves nothing and therefore does not appear in the figure.

The frequency domain synthesis requires three loops. The outer loop runs through the Log2N stages (i.e., each level in Fig. 12-2, starting from the bottom and moving to the top). The middle loop moves through each of the individual frequency spectra in the stage being worked on (i.e., each of the boxes on any one level in Fig. 12-2). The innermost loop uses the butterfly to calculate the points in each frequency spectra (i.e., looping through the samples inside any one box in Fig. 12-2). The overhead boxes in Fig. 12-7 determine the beginning and ending indexes for the loops, as well as calculating the sinusoids needed in the butterflies. Now we come to the heart of this chapter, the actual FFT programs.

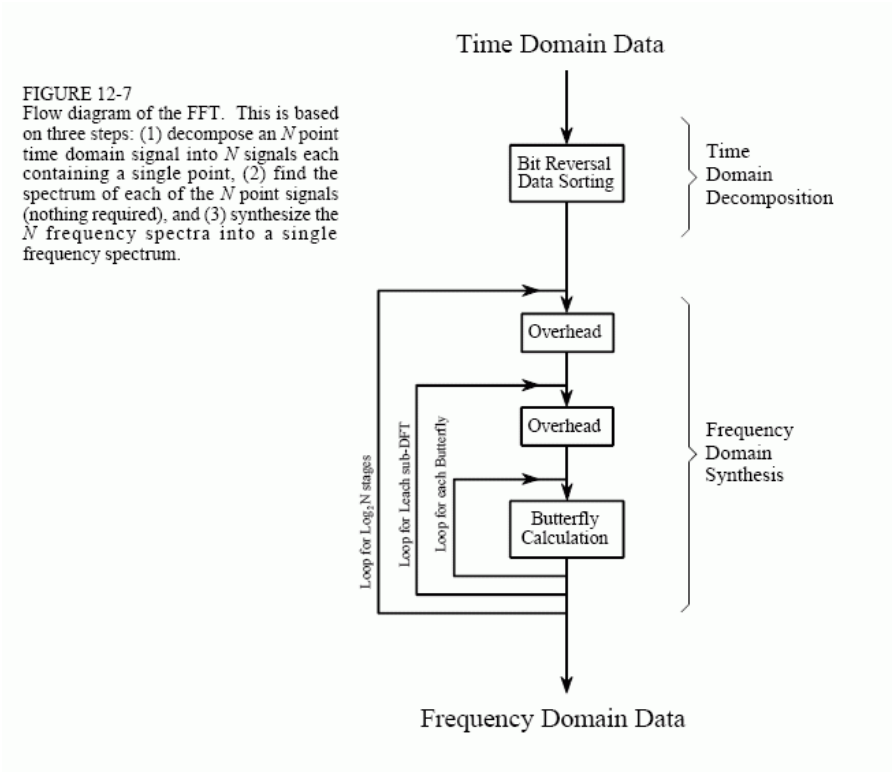
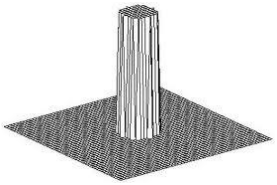


FIGURE 12-7
Flow diagram of the FFT. This is based on three steps: (1) decompose an N point time domain signal into N signals each containing a single point, (2) find the spectrum of each of the N point signals (nothing required), and (3) synthesize the N frequency spectra into a single frequency spectrum.

Filters:

1) Ideal Low pass Filter

A low-pass filter is an electronic filter that passes low-frequency signals and attenuates (reduces the amplitude of) signals with frequencies higher than the cutoff frequency. The actual amount of attenuation for each frequency varies from filter to filter. It is sometimes called a high-cut filter, or treble cut filter when used in audio applications. A low-pass filter is the opposite of a high-pass filter.

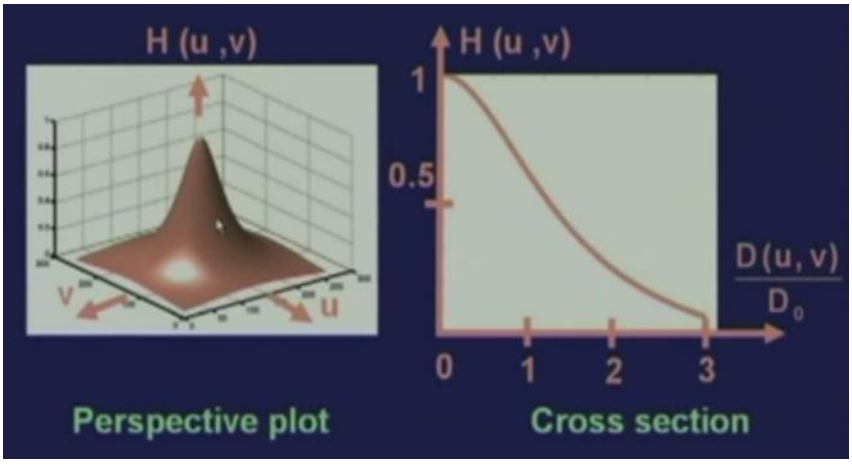


2) Ideal High pass Filter

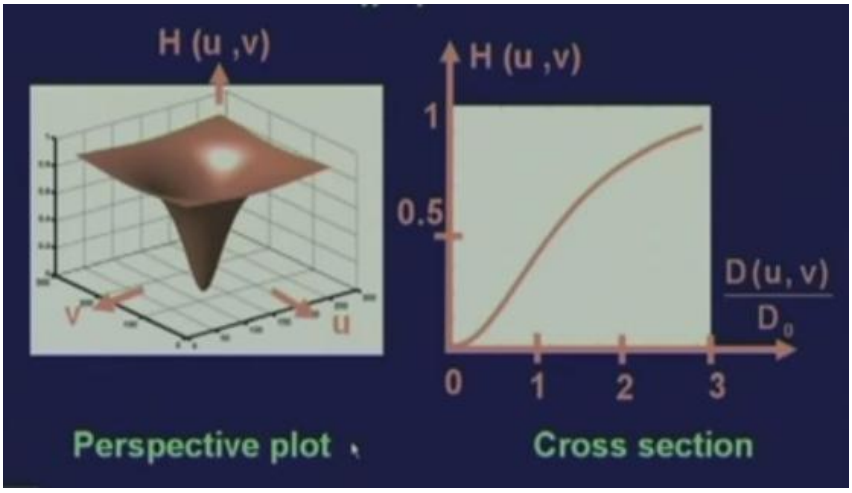
A high-pass filter (HPF) is an electronic filter that passes high-frequency signals but attenuates (reduces the amplitude of) signals with frequencies lower than the cutoff frequency. The actual amount of attenuation for each frequency varies from filter to filter. A high-pass filter is usually modeled as a linear time-invariant system. It is sometimes called a low-cut filter or bass-cut filter. High-pass filters have many uses, such as blocking DC from circuitry sensitive to non-zero average voltages or RF devices. They can also be used in conjunction with a low-pass filter to make a bandpass filter.

3) Butterworth Filter

The frequency response of the Butterworth filter is maximally flat (i.e. has no ripples) in the passband and rolls off towards zero in the stopband. When viewed on a logarithmic Bode plot the response slopes off linearly towards negative infinity. A first-order filter's response rolls off at −6 dB per octave (−20 dB per decade) (all first-order lowpass filters have the same normalized frequency response). A second-order filter decreases at −12 dB per octave, a third-order at −18 dB and so on. Butterworth filters have a monotonically changing magnitude function with ω, unlike other filter types that have non-monotonic ripple in the passband and/or the stopband.



Butterworth LPF



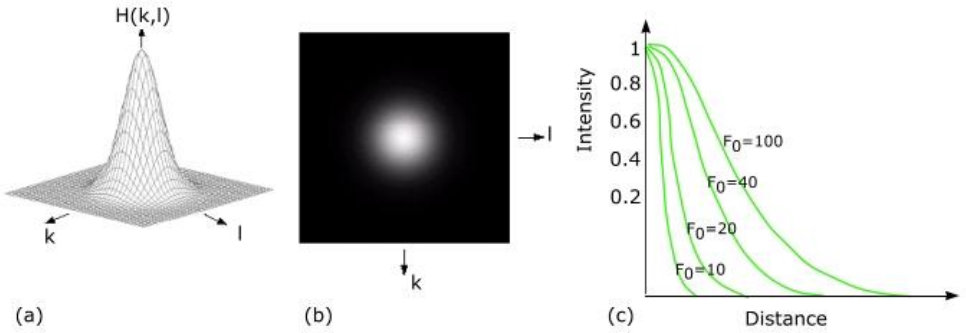
Butterworth HPF

4) Gaussian Filters

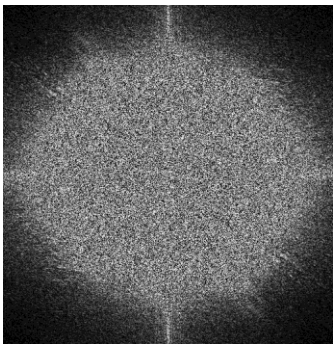
In electronics and signal processing, a Gaussian filter is a filter whose impulse response is a Gaussian function. Gaussian filters are designed to give no overshoot to a step function input while minimizing the rise and fall time. This behavior is closely connected to the fact that the Gaussian filter has the minimum possible group delay. Mathematically, a Gaussian filter modifies the input signal by convolution with a Gaussian function; this transformation is also known as the Weierstrass transform.

In two dimensions, it is the product of two such Gaussians, one per direction:

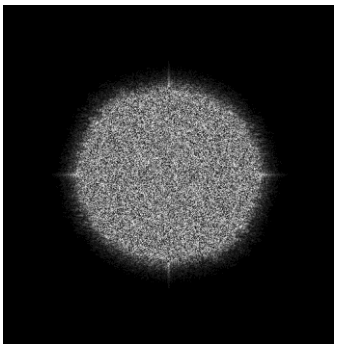
$$g(x,y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$



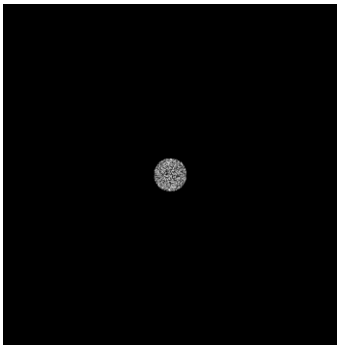
Results:



Butterworth low pass



Gaussian low pass



Ideal low pass

Reference pic

Ideal low pass, Mask 50, Order 3

Ideal high pass, Mask 30, Order 1 (Visible ripples)

Gaussian low pass, Mask 50, Order 3

Butterworth high pass, Mask 3, Order 10 (Cleaner textures)

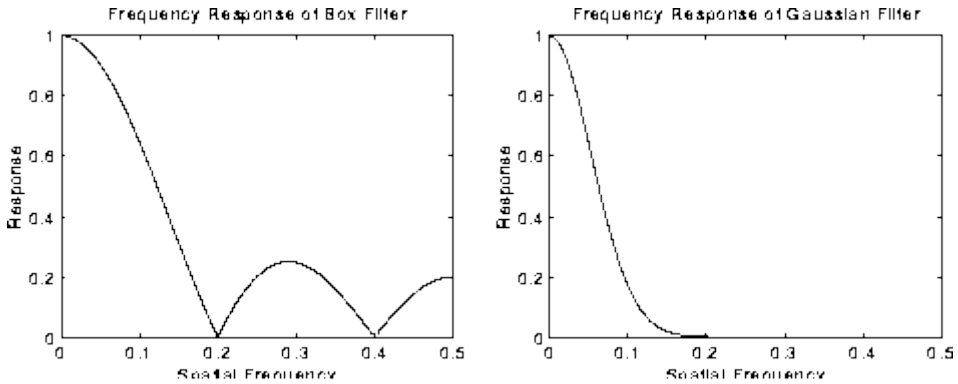
Gaussian high pass, Mask 5, Order 20 (Plastic wrapping effect)

Observation and Inference:

The effect of Gaussian smoothing is to blur an image, in a similar fashion to the mean filter. The degree of smoothing is determined by the standard deviation of the Gaussian. (Larger standard deviation Gaussians, of course, require larger convolution kernels in order to be accurately represented.)

The Gaussian outputs a `weighted average' of each pixel's neighborhood, with the average weighted more towards the value of the central pixels. This is in contrast to the mean filter's uniformly weighted average. Because of this, a Gaussian provides gentler smoothing and preserves edges better than a similarly sized mean filter.

One of the principle justifications for using the Gaussian as a smoothing filter is due to its *frequency response*. Most convolution-based smoothing filters act as lowpass frequency filters. This means that their effect is to remove high spatial frequency components from an image. The frequency response of a convolution filter, *i.e.* its effect on different spatial frequencies, can be seen by taking the Fourier transform of the filter. Figure 5 shows the frequency responses of a 1-D mean filter with width 5 and also of a Gaussian filter with σ = 3.



Frequency responses of Box (*i.e.* mean) filter (width 5 pixels) and Gaussian filter (σ = 3 pixels). The spatial frequency axis is marked in cycles per pixel, and hence no value above 0.5 has a real meaning.

Both filters attenuate high frequencies more than low frequencies, but the mean filter exhibits oscillations in its frequency response. The Gaussian on the other hand shows no oscillations. In fact, the shape of the frequency response curve is itself (half a) Gaussian. So by choosing an appropriately sized Gaussian filter we can be fairly confident about what range of spatial frequencies are still present in the image after filtering, which is not the case of the mean filter. This has consequences for some edge detection techniques, as mentioned in the section on zero crossings. (The Gaussian filter also turns out to be very similar to the optimal smoothing filter for edge detection under the criteria used to derive the Canny edge detector which uses the Gaussian HPF as a base filter.)

The complexity or filter type is defined by the filters "order", and which is dependant upon the number of reactive components such as capacitors or inductors within its design. We also know that the rate of roll-off and therefore the width of the transition band, depends upon the order number of the filter and that for a simple first-order filter it has a standard roll-off rate of 20dB/decade or 6dB/octave. Then, for a filter that has an nth number order, it will have a subsequent roll-off rate of 20n dB/decade or 6N dB/octave. So a first-order filter has a roll-off rate of 20dB/decade (6dB/octave), a second-order filter has a roll-off rate of 40dB/decade (12dB/octave), and a fourth-order filter has a roll-off rate of 80dB/decade (24dB/octave), etc. High-order filters, such as third, fourth, and fifth-order are usually

formed by cascading together single first-order and second-order filters. For example, two second-order low pass filters can be cascaded together to produce a fourth-order low pass filter, and so on. Although there is no limit to the order of the filter that can be formed, as the order increases so does its size and cost, also its accuracy declines.