- 10. Write a function to implement image enlargement using zero-interleaving and spatial filtering. The function should have the syntax

```
imenlarge(image,n,filt)
```

where `n` is the number of times the interleaving is to be done, and `filt` is the filter to use. So, for example, the command

```
imenlarge(head,2,bfilt);
```

would enlarge an image to four times its size, using the 5 × 5 filter described in Section 6.4.

7 The Fourier Transform

7.1 Introduction

The Fourier Transform is of fundamental importance to image processing. It allows us to perform tasks that would be impossible to perform any other way; its efficiency allows us to perform other tasks more quickly. The Fourier Transform provides, among other things, a powerful alternative to linear spatial filtering; it is more efficient to use the Fourier Transform than a spatial filter for a large filter. The Fourier Transform also allows us to isolate and process particular image "frequencies," and so perform low pass and high pass filtering with a great degree of precision.

Before we discuss the Fourier Transform of images, we shall investigate the one-dimensional Fourier Transform, and a few of its properties.

7.2 Background

Our starting place is the observation that a periodic function may be written as the sum of sines and cosines of varying amplitudes and frequencies. For example, in Figure 7.1 we plot a function and its decomposition into sine functions.

Some functions will require only a finite number of functions in their decomposition; others will require an infinite number. For example, a "square wave," such as is shown in Figure 7.2, has the decomposition

$$f(x) = \sin x + \frac{1}{3}\sin 3x + \frac{1}{5}\sin 5x + \frac{1}{7}\sin 7x + \frac{1}{9}\sin 9x + \cdots \tag{7.1}$$

In Figure 7.2 we take the first five terms only to provide the approximation. The more terms of the series we take, the closer the sum will approach the original function.

This can be formalized; if $f(x)$ is a function of period $2T$, then we can write

$$f(x) = a_0 + \sum_{n=1}^{\infty}\left(a_n \cos\frac{n\pi x}{T} + b_n \sin\frac{n\pi x}{T}\right)$$

Figure 7.1: A function and its trigonometric decomposition

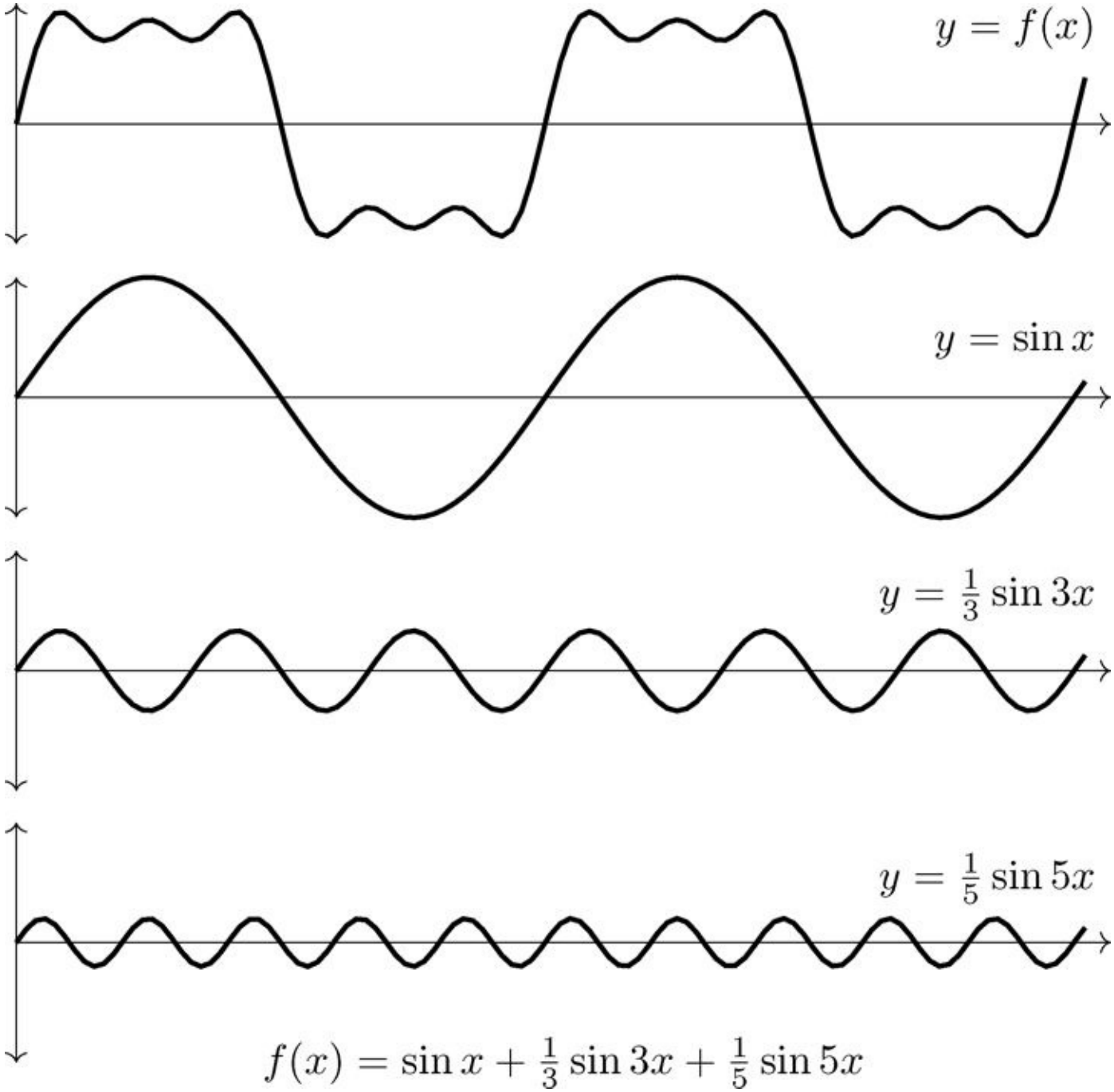
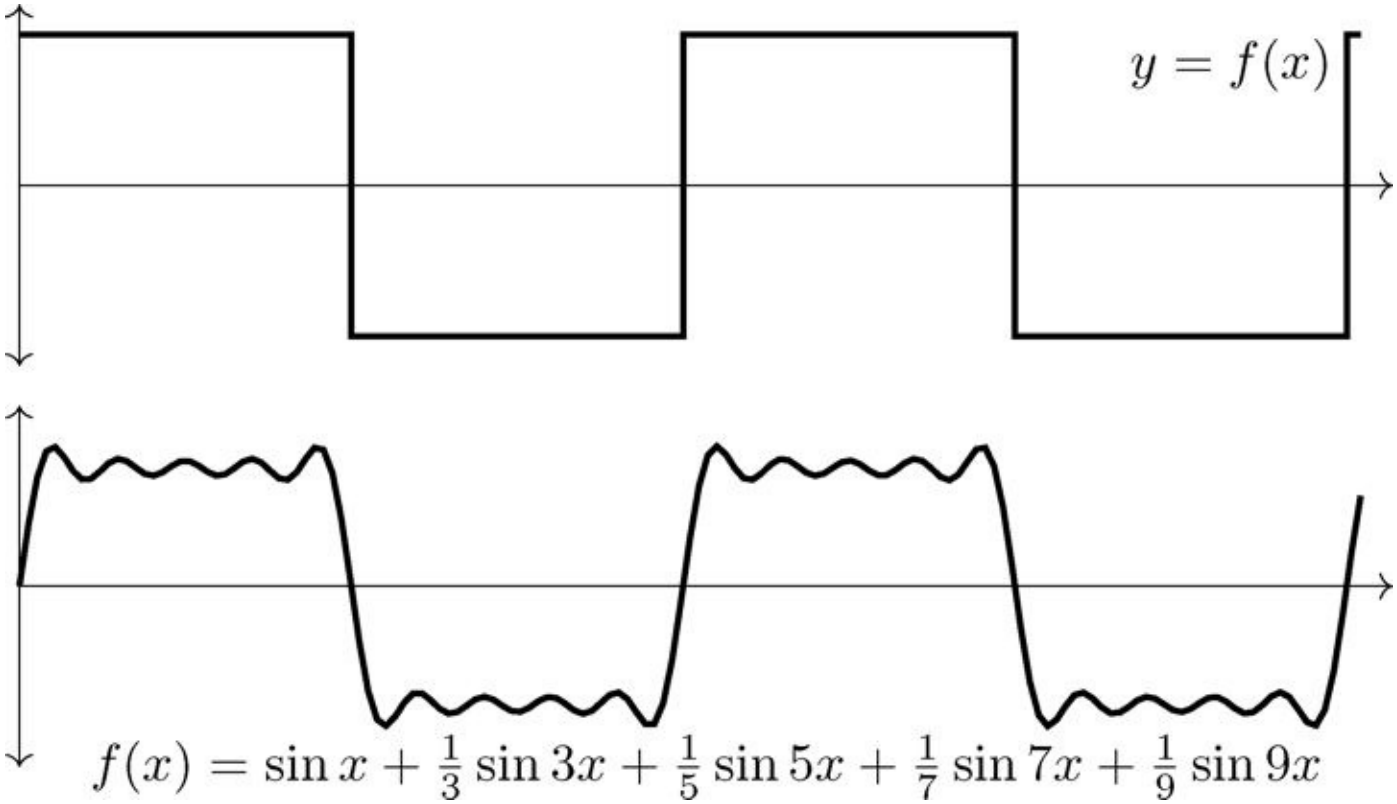


Figure 7.2: A square wave and its trigonometric approximation



where

$$a_0 = \frac{1}{2T}\int_{-T}^{T} f(x)\,dx$$
$$a_n = \frac{1}{T}\int_{-T}^{T} f(x)\cos\frac{n\pi x}{T}\,dx, \quad n = 1, 2, 3, \ldots$$
$$b_n = \frac{1}{T}\int_{-T}^{T} f(x)\sin\frac{n\pi x}{T}\,dx, \quad n = 1, 2, 3, \ldots$$

These are the equations for the *Fourier series expansion* of *f*(*x*), and they can be expressed in complex form:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n \exp\Big(\frac{in\pi x}{T}\Big)dx$$

where

$$c_n = \frac{1}{2T}\int_{-T}^{T} f(x)\exp\Big(\frac{-in\pi x}{T}\Big)dx.$$

If the function is non-periodic, we can obtain similar results by letting $T \to \infty$, in which case

$$f(x) = \int_0^{\infty} [a(\omega)\cos\omega x + b(\omega)\sin\omega x]d\omega$$

where

$$a(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} f(x) \cos \omega x \, dx,$$
$$b(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} f(x) \sin \omega x \, dx.$$

These equations can be written again in complex form:

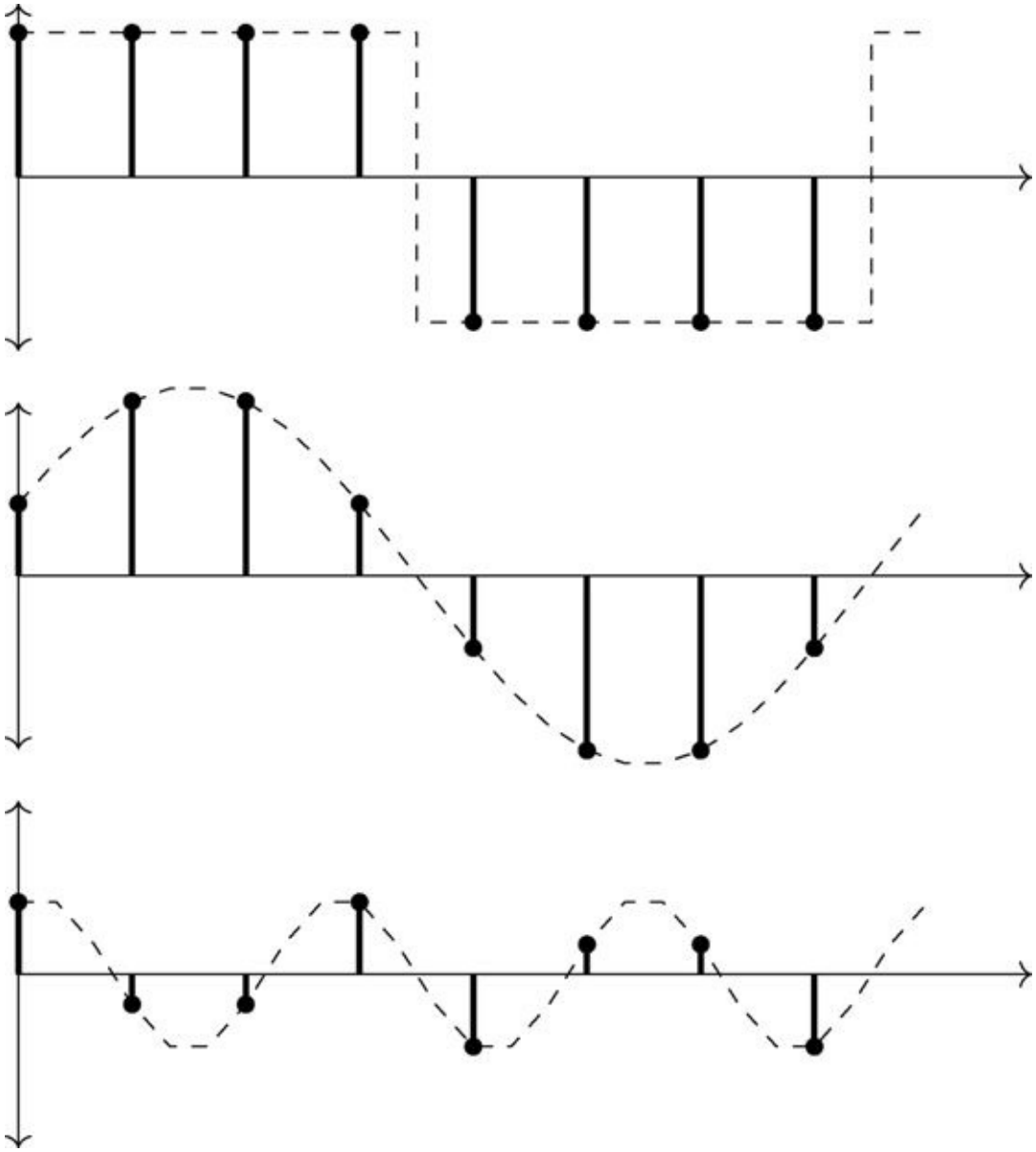
$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{i\omega x} d\omega,$$
$$F(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} f(x) e^{i\omega x} dx.$$

In this last form, the functions $f(x)$ and $F(\omega)$ form a *Fourier transform pair*. Further details can be found, for example, in James [23].

7.3 The One-Dimensional Discrete Fourier Transform

When we deal with a *discrete* function, as we shall do for images, the situation from the previous section changes slightly. Since we only have to obtain a finite number of values, we only need a finite number of functions to do it.

Figure 7.3: Expressing a discrete function as the sum of sines



Consider, for example the discrete sequence

$$1, \quad 1, \quad 1, \quad 1, \quad -1, \quad -1, \quad -1, \quad -1$$

which we may take as a discrete approximation to the square wave of Figure 7.2. This can be expressed as the sum of only *two* sine functions; this is shown in Figure 7.3. We shall see below how to obtain those sequences.

The Fourier Transform allows us to obtain those individual sine waves that compose a given function or sequence. Since we shall be concerned with discrete sequences, and of course images, we shall investigate only the *discrete Fourier Transform*, abbreviated DFT.

Definition of the One-Dimensional DFT

Suppose

$$\mathbf{f} = [f_0, f_1, f_2, \ldots, f_{N-1}]$$

is a sequence of length N. We define its *discrete Fourier Transform* to be the sequence

$$\mathbf{F} = [F_0, F_1, F_2, \ldots, F_{N-1}]$$

where

$$F_u = \frac{1}{N} \sum_{x=0}^{N-1} \exp\left[-2\pi i \frac{xu}{N}\right] f_x. \tag{7.2}$$

Note the similarity between this equation and the equations for the Fourier series expansion discussed in the previous section. Instead of an integral, we now have a finite sum. This definition can be expressed as a matrix multiplication:

$$F = \mathcal{F} f$$

where F is an N x N matrix defined by

$$\mathcal{F}_{m,n} = \frac{1}{N} \exp\left[-2\pi i \frac{mn}{N}\right].$$

Given N, we shall define

$$\omega = \exp\left[\frac{-2\pi i}{N}\right]$$

so that

$$\mathcal{F}_{m,n} = \frac{1}{N} \omega^{mn}.$$

Then we can write

$$\mathcal{F} = \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \omega^{12} & \cdots & \omega^{3(N-1)} \\ 1 & \omega^4 & \omega^8 & \omega^{12} & \omega^{16} & \cdots & \omega^{4(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \omega^{4(N-1)} & \cdots & \omega^{(N-1)^2} \end{bmatrix}$$

Example. Suppose **f** = [1, 2, 3, 4] so that $N = 4$. Then

$$\begin{aligned} \omega &= \exp\left[\frac{-2\pi i}{4}\right] \\ &= \exp\left[-\frac{\pi i}{2}\right] \\ &= \cos\left(-\frac{\pi}{2}\right) + i \sin\left(-\frac{\pi}{2}\right) \\ &= -i. \end{aligned}$$

Then we have

$$\mathcal{F} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & (-i)^2 & (-i)^3 \\ 1 & (-i)^2 & (-i)^4 & (-i)^6 \\ 1 & (-i)^3 & (-i)^6 & (-i)^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$$

and so

$$\mathbf{F} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 10 \\ -2+2i \\ -2 \\ -2-2i \end{bmatrix}.$$

The Inverse DFT

The formula for the inverse DFT is very similar to the forward transform:

$$x_u = \sum_{x=0}^{N-1} \exp\left[2\pi i \frac{xu}{N}\right] F_u. \tag{7.3}$$

If you compare this equation with Equation 7.2, you will see that there are really only two differences:

1. There is no scaling factor $1/N$
2. The sign inside the exponential function has been changed to positive

As with the forward transform, we can express this as a matrix product:

$$f = \mathcal{F}^{-1} F$$

with

$$\mathcal{F}^{-1} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \overline{\omega}^1 & \overline{\omega}^2 & \overline{\omega}^3 & \overline{\omega}^4 & \cdots & \overline{\omega}^{N-1} \\ 1 & \overline{\omega}^2 & \overline{\omega}^4 & \overline{\omega}^6 & \overline{\omega}^8 & \cdots & \overline{\omega}^{2(N-1)} \\ 1 & \overline{\omega}^3 & \overline{\omega}^6 & \overline{\omega}^9 & \overline{\omega}^{12} & \cdots & \overline{\omega}^{3(N-1)} \\ 1 & \overline{\omega}^4 & \overline{\omega}^8 & \overline{\omega}^{12} & \overline{\omega}^{16} & \cdots & \overline{\omega}^{4(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \overline{\omega}^{N-1} & \overline{\omega}^{2(N-1)} & \overline{\omega}^{3(N-1)} & \overline{\omega}^{4(N-1)} & \cdots & \overline{\omega}^{(N-1)^2} \end{bmatrix}$$

where

$$\overline{\omega} = \frac{1}{\omega} = \exp\left[\frac{2\pi i}{N}\right].$$

In MATLAB or Octave, we can calculate the forward and inverse transforms with `fft` and `ifft`. Here `fft` stands for *Fast Fourier Transform*, which is a fast and efficient method of performing the DFT (see below for details). For example:

```
a =

     1     2     3     4     5     6

>> fft(a')

ans =

  21.0000
  -3.0000 + 5.1962i
  -3.0000 + 1.7321i
  -3.0000
  -3.0000 - 1.7321i
  -3.0000 - 5.1962i
```

MATLAB/Octave

We note that to apply a DFT to a single vector in MATLAB or Octave, we should use a column vector.

In Python it is very similar; there are modules for the DFT in both the `scipy` and `numpy` libraries. For ease of use, we can first import all the functions we need from `numpy.fft`, and then apply them:

```
In:  from numpy.fft import *
In:  a = range(1,7)
In:  L = fft(a)
In:  for x in L: print '%0.4f%␣+0.4fi' % (x.real, x.imag)

21.0000 +0.0000i
-3.0000 +5.1962i
-3.0000 +1.7320i
-3.0000 +0.0000i
-3.0000 -1.7321i
-3.0000 -5.1961i
```

Python

7.4 Properties of the One-Dimensional DFT

The one-dimensional DFT satisfies many useful and important properties. We will investigate some of them here. A more complete list is given, for example, by Jain [22].

Linearity. This is a direct consequence of the definition of the DFT as a matrix product. Suppose `f` and `g` are two vectors of equal length, and p and q are scalars, with `h` = p`f` + q`g`. If `F, G`, and `H` are the DFTs of `f, g`, and `h`, respectively, we have

$$\mathtt{H} = p\mathtt{F} + q\mathtt{G}.$$

This follows from the definitions of

$$\mathtt{F} = \mathcal{F}\mathtt{f}, \quad \mathtt{G} = \mathcal{F}\mathtt{g}, \quad \mathtt{H} = \mathcal{F}\mathtt{h}$$

and the linearity of the matrix product.

Shifting. Suppose we multiply each element x_n of a vector x by $(-1)^n$. In other words, we change the sign of every second element. Let the resulting vector be denoted x'. The the DFT X' of x' is equal to the DFT X of x with the swapping of the left and right halves.

Let's do a quick example:

```
>> x = [2 3 4 5 6 7 8 1];

>> x1=(-1).^[0:7].*x

x1 =

     2    -3     4    -5     6    -7     8    -1

>> X=fft(x')

  36.0000
  -9.6569 + 4.0000i
  -4.0000 - 4.0000i
   1.6569 - 4.0000i
   4.0000
   1.6569 + 4.0000i
  -4.0000 + 4.0000i
  -9.6569 - 4.0000i

>> X1=fft(x1')

X1 =

   4.0000
   1.6569 + 4.0000i
  -4.0000 + 4.0000i
  -9.6569 - 4.0000i
  36.0000
  -9.6569 + 4.0000i
  -4.0000 - 4.0000i
   1.6569 - 4.0000i
```

MATLAB/Octave

Notice that the first four elements of X are the last four elements of X1, and vice versa. Note that in Python the above example could be obtained with:

```
In:  x = np.array([2, 3, 4, 5, 6, 7, 8, 1])
In:  x1 = copy(x)
In:  x1[1::2] = x1[1::2]*-1
In:  L = fft(x)
In:  L1 = fft(x1)
```

Python

and then the arrays `L` and `L1` can be displayed using the "print" example above.

Conjugate symmetry If `x` is real, and of length N, then its DFT `X` satisfies the condition that

$$X_k = \overline{X_{N-k}},$$

where $\overline{X_{N-k}}$ is the complex conjugate of X_{N-k}, for all $k = 1, 2, 3, \ldots, N-1$. So in our example of length 8, we have

$$X_1 = \overline{X_7}, \quad X_2 = \overline{X_6}, \quad X_3 = \overline{X_5}.$$

In this case, we also have $X_4 = \overline{X_4}$, which means X_4 must be real. In fact, if N is even, then $X_{N/2}$ will be real. Examples can be seen above.

Convolution. Suppose `x` and `y` are two vectors of the same length N. Then we define their *convolution* (or, more properly, their *circular convolution*) to be the vector

$$z = x * y$$

where

$$z_k = \frac{1}{n} \sum_{n=0}^{N-1} x_n y_{k-n}.$$

For example, if $N = 4$, then

$$\begin{aligned}
z_0 &= \frac{1}{4}(x_0y_0 + x_1y_{-1} + x_2y_{-2} + x_3y_{-3}) \\
z_1 &= \frac{1}{4}(x_0y_1 + x_1y_0 + x_2y_{-1} + x_3y_{-2}) \\
z_2 &= \frac{1}{4}(x_0y_2 + x_1y_1 + x_2y_0 + x_3y_{-1}) \\
z_3 &= \frac{1}{4}(x_0y_3 + x_2y_2 + x_2y_1 + x_3y_0)
\end{aligned}$$

The negative indices can be interpreted by imagining the `y` vector to be periodic, and can be indexed backward from 0 as well as forward:

$$\begin{array}{rccccccccc}
 & \cdots & y_0 & y_1 & y_2 & y_3 & y_0 & y_1 & y_2 & y_3 & \cdots \\
= & \cdots & y_0 & y_{-3} & y_{-2} & y_{-1} & y_0 & y_1 & y_2 & y_3 & \cdots
\end{array}$$

Thus, $y_{-1} = y_3$, $y_{-2} = y_2$ and $y_{-3} = y_1$. It can be checked from the definition that convolution is *commutative* (the order of the operands is irrelevant):

$$x * y = y * x.$$

One way of thinking about circular convolution is by a "sliding" operation. Consider the array x against an array consisting of the array y backwards twice: Sliding the x array into different positions and multiplying the corresponding elements and adding will produce the result. This is shown in Figure 7.4.

Circular convolution as defined looks like a messy operation. However, it can be defined in terms of polynomial products. Suppose $p(u)$ is the polynomial in u whose coefficients are the elements of `x`. Let $q(u)$ be the polynomial whose coefficients are the elements of `y`. Form the product $p(u)q(u)(1+u^N)$, and extract the coefficients of u^N to u^{2N-1}. These will be our required circular convolution.

For example, suppose we have:

$$x = [1, \quad 2, \quad 3, \quad 4], \qquad y = [5, \quad 6, \quad 7, \quad 8].$$

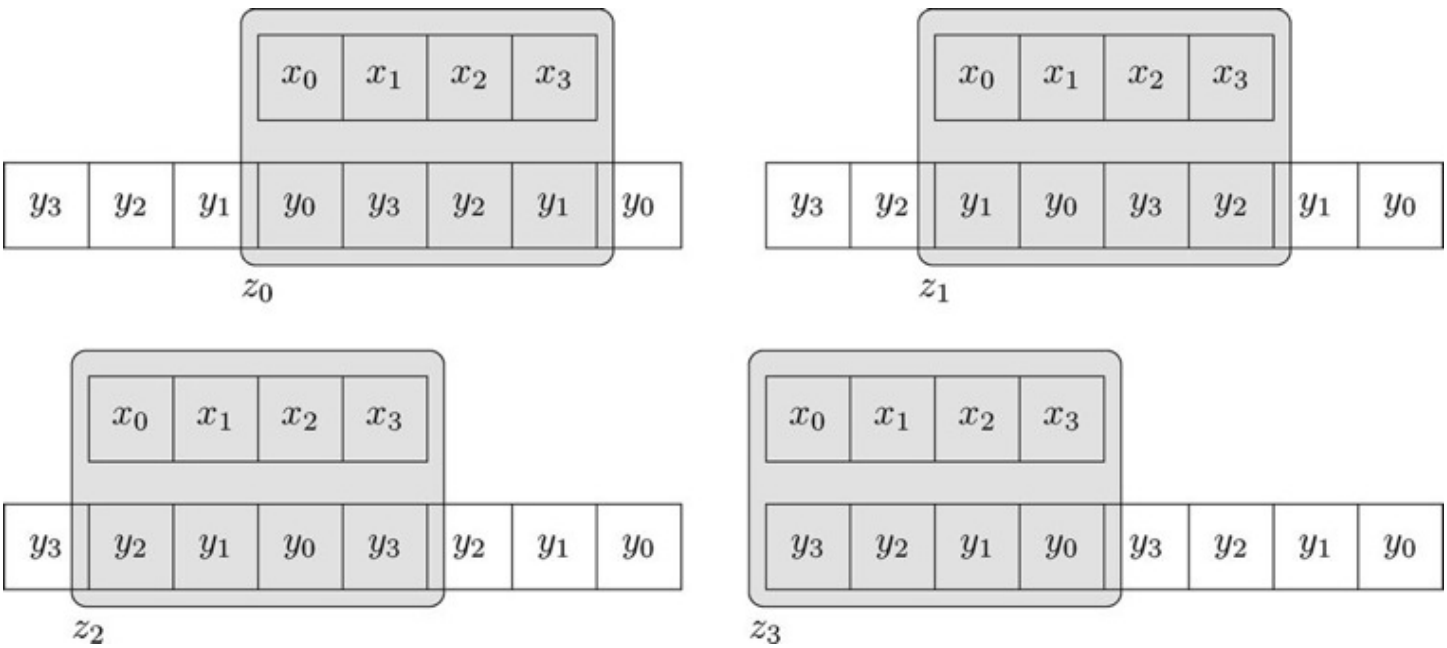
Then we have

$$p(u) = 1 + 2u + 3u^2 + 4u^3$$

and

$$q(u) = 5 + 6u + 7u^2 + 8u^3.$$

Figure 7.4: Visualizing circular convolution



Then we expand

$$p(u)q(u)(1+u^4) = 5+16x+34u^2+60u^3+66u^4+68u^5+66u^6+60u^7+61u^8+52u^9+32u^{10}.$$

Extracting the coefficients of u^4, u^5, ..., u^7 we obtain

$$x * y = [66, \quad 68, \quad 66, \quad 60].$$

MATLAB has a `conv` function, which produces the coefficients of the polynomial $p(u)q(u)$ as defined above:

```
>> a = [1 2 3 4]

a =

     1     2     3     4

>> b = [5 6 7 8]

b =

     5     6     7     8

>> conv(a,b)

ans =

     5    16    34    60    61    52    32
```

MATLAB/Octave

To perform circular convolution, we first notice that the polynomial $p(u)(1 + u^N)$ can be obtained by simply repeating the coefficients of `x`. So, given the above two arrays:

```
>> N = length(a);
>> C = conv(a,[b,b])
>> C(N:2*N-1)
ans =

   60   66   68   66
```

MATLAB/Octave

which is exactly what we obtained above. Python works very similarly, using the command `convolve`:

`convolve`:

```
In:  a = [1,2,3,4]
In:  b = [5,6,7,8]
In:  convolve(a,b)
Out: array([ 5, 16, 34, 60, 61, 52, 32])
In:  convolve(a,b+b,'valid')[:-1]
Out: array([60, 66, 68, 66])
```

Python

The importance of convolution is the *convolution theorem*, which states:

> Suppose `x` and `y` are vectors of equal length. Then the DFT of their circular convolution $x * y$ is equal to the element-by-element product of the DFTs of `x` and `y`.

So if `Z, X, Y` are the DFTs of `z = x * y, x` and `y`, respectively, then

$$Z = X.Y.$$

We can check this with our vectors above:

```
>> C = conv(a,[b,b]);
>> fft(C')


ans =

   1.0e+02 *

   2.6000
        0 - 0.0800i
   0.0400
        0 + 0.0800i

>> fft(a').*fft(b');

ans =

   1.0e+02 *

   2.6000
        0 - 0.0800i
   0.0400
        0 + 0.0800i
```

MATLAB/Octave

The previous computations can be done in Python, but using numpy arrays instead of lists. The `convolve` function here from `numpy` acts only on one-dimensional arrays, and the `hstack` function concatenates two arrays horizontally:

```
In:  a = np.array([1,2,3,4])
In:  b = np.array([5,6,7,8])
In:  C = np.convolve(a,np.hstack([b,b]),'valid')[:-1]
In:  fft(C)
Out: array([ 260.+0.j,   -8.+0.j,   -4.+0.j,   -8.+0.j])
In:  fft(a)*fft(b)
Out: array([ 260.+0.j,    0.-8.j,    4.-0.j,    0.+8.j])
```

Python

Note that in each case the results are the same. The convolution theorem thus provides us with another way of performing convolution: multiply the DFTs of our two vectors and invert the result:

```
>> fft(a').*fft(b');
>> ifft(ans)'

ans =

    66    68    66    60
```

MATLAB/Octave

or

```
In:  ab = fft(a)*fft(b)
In:  real(ifft(ab))
Out: array([66., 68., 66., 60.])
```

Python

A formal proof of the convolution theorem for the DFT is given by Petrou [34].

The Fast Fourier Transform. One of the many aspects that make the DFT so attractive for image processing is the existence of very fast algorithms to compute it. There are a number of extremely fast and efficient algorithms for computing a DFT; such an algorithm is called a *Fast Fourier Transform*, or FFT. The use of an FFT vastly reduces the time needed to compute a DFT.

One FFT method works recursively by dividing the original vector into two halves, computing the FFT of each half, and then putting the results together. This means that the FFT is most efficient when the vector length is a power of 2. This method is discussed in Appendix C.

Table 7.1 shows that advantage gained by using the FFT algorithm as opposed to the direct arithmetic definition of Equations 7.6 and 7.7 by comparing the number of multiplications required for each method. For a vector of length 2^n, the direct method takes $(2^n)^2 = 2^{2n}$ multiplications; the FFT only $n2^n$. The saving in time is thus of an order of $2^n/n$. Clearly the advantage of using an FFT algorithm becomes greater as the size of the vector increases.

Because of the computational advantage, any implementation of the DFT will use an FFT algorithm.

Table 7.1 Comparison of FFT and direct arithmetic

2^n	Direct arithmetic	FFT	Increase in speed
4	16	8	2.0
8	84	24	2.67
16	256	64	4.0

2	Direct arithmetic	FFT	Increase in speed
32	1024	160	6.4
64	4096	384	10.67
128	16384	896	18.3
256	65536	2048	32.0
512	262144	4608	56.9
1024	1048576	10240	102.4

7.5 The Two-Dimensional DFT

In two dimensions, the DFT takes a matrix as input, and returns another matrix, of the same size, as output. If the original matrix values are $f(x, y)$, where x and y are the indices, then the output matrix values are $F(u, v)$. We call the matrix F the *Fourier Transform of f* and write

$$F = \mathcal{F}(f).$$

Then the original matrix f is the *inverse Fourier Transform of F*, and we write

$$f = \mathcal{F}^{-1}(F).$$

We have seen that a (one-dimensional) function can be written as a sum of sines and cosines. Given that an image may be considered a two-dimensional function $f(x, y)$, it seems reasonable to assume that f can be expressed as sums of "corrugation" functions which have the general form

$$z = a \sin(bx + cy).$$

A sample such function is shown in Figure 7.5. And this is in fact exactly what the two-dimensional Fourier Transform does: it rewrites the original matrix in terms of sums of corrugations. The amplitude and period of each corrugation defines its position on the Fourier spectrum, as shown in Figure 7.6. From Figure 7.6 we note that the more spread out the corrugation, the closer to the center of the spectrum it will be positioned. This is because a spread out corrugation means large values of $1/x$ and $1/y$, hence small values of x and y. Similarly, "squashed" corrugations (with small values of $1/x$ and $1/y$) will be positioned further from the center.

The definition of the two-dimensional discrete Fourier Transform is very similar to that for one dimension. The forward and inverse transforms for an M x N matrix, where for notational convenience we assume that the x indices are from 0 to $M - 1$ and the y indices are from 0 to $N - 1$, are:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \exp\left[-2\pi i \left(\frac{xu}{M} + \frac{yv}{N}\right)\right]. \tag{7.4}$$

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) \exp\left[2\pi i \left(\frac{xu}{M} + \frac{yv}{N}\right)\right]. \tag{7.5}$$

These are horrendous looking formulas, but if we spend a bit of time pulling them apart, we shall see that they aren't as bad as they look.

Before we do this, we note that the formulas given in Equations 7.4 and 7.5 are not used by all authors. The main change is the position of the scaling factor 1/*MN*. Some people put it in front of the sums in the forward formula. Others put a factor of $1/\sqrt{MN}$ in front of both sums. The point is the sums by themselves would produce a result (after both forward and inverse transforms) which is too large by a factor of *MN*. So somewhere in the forward-inverse formulas a corresponding 1/*MN* must exist; it doesn't really matter where.

Some Properties of the Two-Dimensional Fourier Transform

All the properties of the one-dimensional DFT transfer into two dimensions. But there are some further properties not previously mentioned, which are of particular use for image processing.

Similarity. First notice that the forward and inverse transforms are very similar, with the exception of the scale factor 1/*MN* in the inverse transform, and the negative sign in the exponent of the forward transform. This means that the same algorithm, only very slightly adjusted, can be used for both the forward and inverse transforms.

The DFT as a spatial filter. Note that the values

$$\exp\left[\pm 2\pi i \left(\frac{xu}{M} + \frac{yv}{N}\right)\right]$$

are independent of the values *f* or *F*. This means that they can be calculated in advance, and only then put into the formulas above. It also means that every value *F*(*u, v*) is obtained by multiplying every value of *f*(*x, y*) by a fixed value, and adding up all the results. But this is precisely what a linear spatial filter does: it multiplies all elements under a mask with fixed values, and adds them all up. Thus, we can consider the DFT as a linear spatial filter which is as big as the image. To deal with the problem of edges, we assume that the image is tiled in all directions, so that the mask always has image values to use.

Figure 7.5: A “corrugation” function

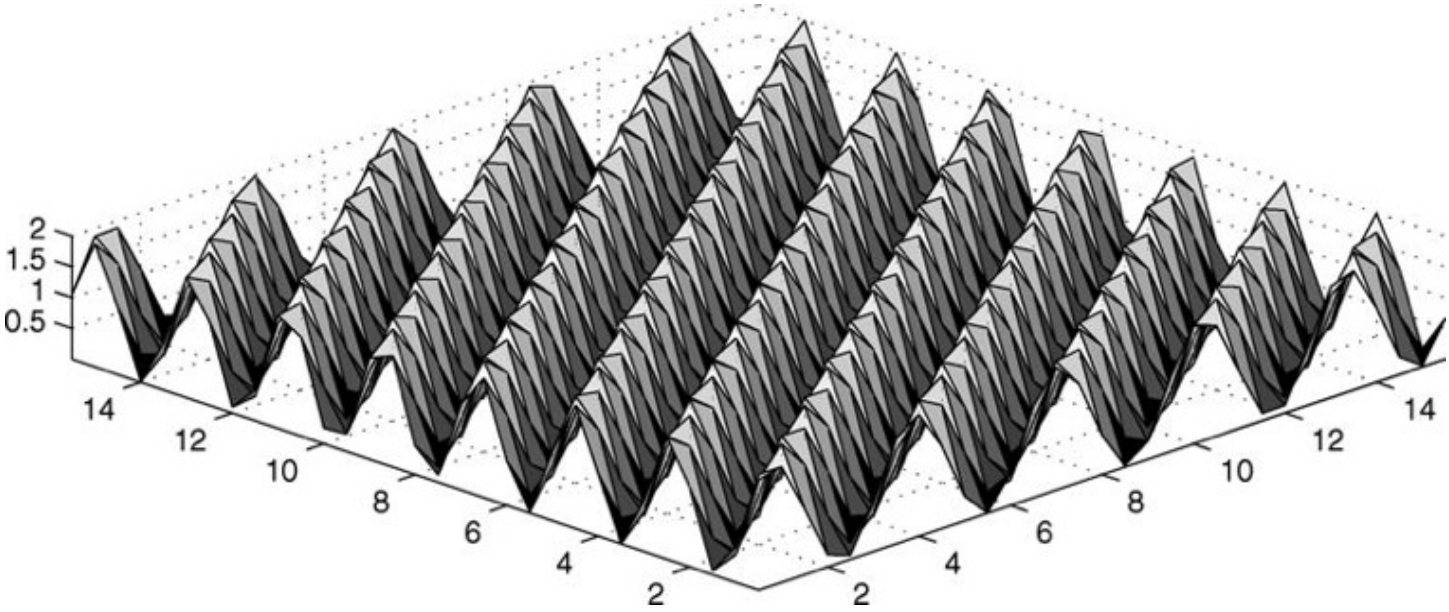
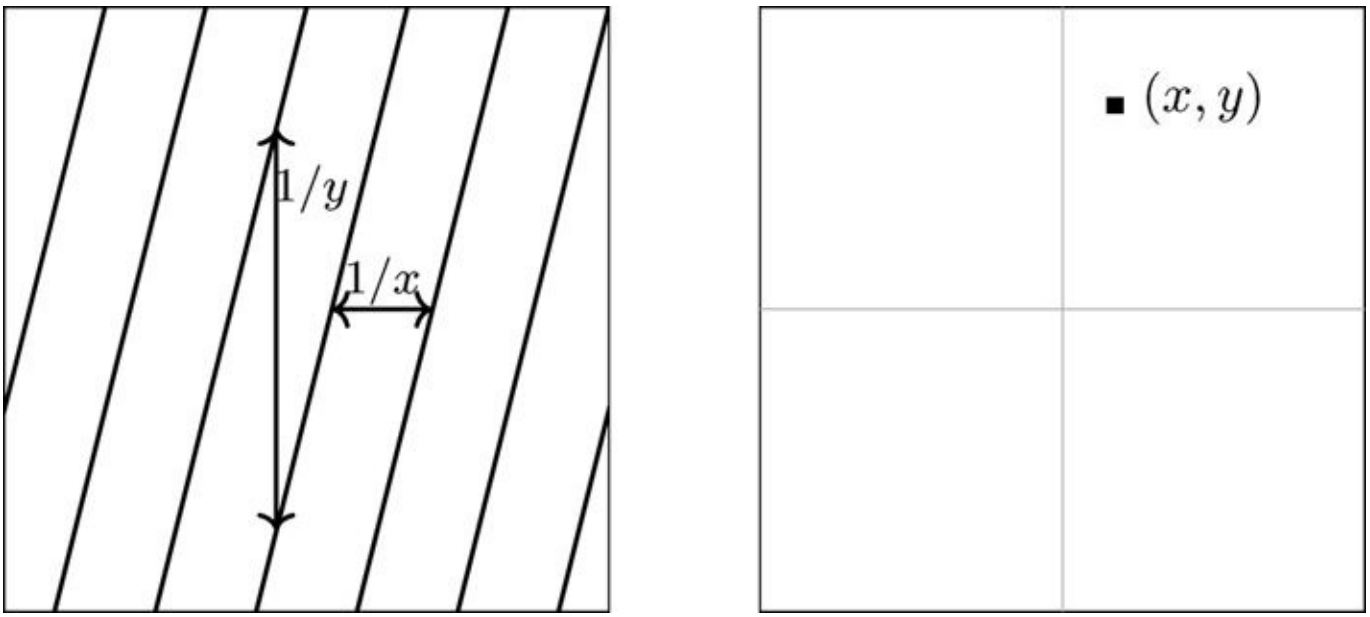


Figure 7.6: Where each corrugation is positioned on the spectrum



Separability. Notice that the Fourier Transform “filter elements” can be expressed as products:

$$\exp\left[2\pi i\left(\frac{xu}{M}+\frac{yv}{N}\right)\right]=\exp\left[2\pi i\frac{xu}{M}\right]\exp\left[2\pi i\frac{yv}{N}\right].$$

The first product value

$$\exp\left[2\pi i\frac{xu}{M}\right]$$

depends only on x and u, and is independent of y and v. Conversely, the second product value

$$\exp\left[2\pi i\frac{yv}{N}\right]$$

depends only on y and v, and is independent of x and u. This means that we can break down our formulas above to simpler formulas that work on single rows or columns:

$$F(u) = \sum_{x=0}^{M-1} f(x) \exp\left[-2\pi i \frac{xu}{M}\right], \tag{7.6}$$

$$f(x) = \frac{1}{M} \sum_{u=0}^{M-1} F(u) \exp\left[2\pi i \frac{xu}{M}\right]. \tag{7.7}$$

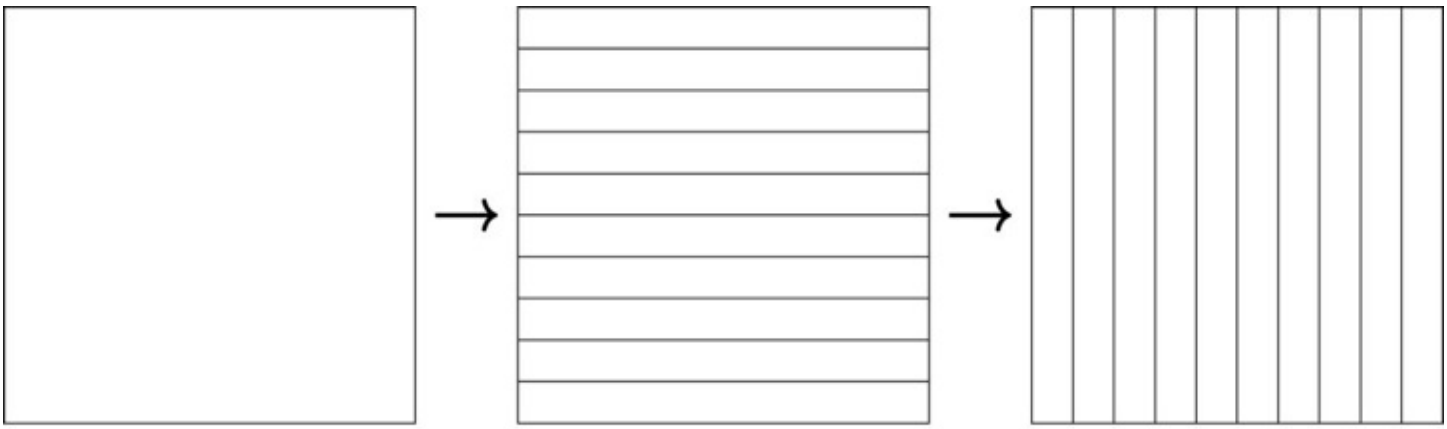
If we replace x and u with y and v, we obtain the corresponding formulas for the DFT of matrix columns. These formulas define the *one-dimensional DFT* of a vector, or simply the DFT.

The 2-D DFT can be calculated by using this property of "separability"; to obtain the 2-D DFT of a matrix, we first calculate the DFT of all the rows, and then calculate the DFT of all the columns of the result, as shown in Figure 7.7. Since a product is independent of the order, we can equally well calculate a 2-D DFT by calculating the DFT of all the columns first, and then calculating the DFT of all the rows of the result.

Linearity An important property of the DFT is its linearity; the DFT of a sum is equal to the sum of the individual DFTs, and the same goes for scalar multiplication:

$$\mathcal{F}(f+g) = \mathcal{F}(f) + \mathcal{F}(g)$$
$$\mathcal{F}(kf) = k\mathcal{F}(f)$$

Figure 7.7: Calculating a 2D DFT



where k is a scalar, and f and g are matrices. This follows directly from the definition given in Equation 7.4.

This property is of great use in dealing with image degradation such as noise, which can be modelled as a sum:

$$d = f + n$$

where f is the original image, n is the noise, and d is the degraded image. Since

$$\mathcal{F}(d) = \mathcal{F}(f) + \mathcal{F}(n)$$

we may be able to remove or reduce n by modifying the transform. As we shall see, some noise appears on the DFT in a way that makes it particularly easy to remove.

The convolution theorem. This result provides one of the most powerful advantages of using the DFT. Suppose we wish to convolve an image M with a spatial filter S. Our method has been place S over each pixel of M in turn, calculate the product of all corresponding gray values of M and elements of S, and add the results. The result is called the *digital convolution* of M and S, and is denoted

$$M * S.$$

This method of convolution can be very slow, especially if S is large. The *convolution theorem* states that the result $M * S$ can be obtained by the following sequence of steps:

1. Pad S with zeros so that it is the same size as M; denote this padded result by S'.
2. Form the DFTs of both M and S, to obtain $F(M)$ and $F(S')$.
3. Form the element-by-element product of these two transforms: $\mathcal{F}(M) \cdot \mathcal{F}(S')$.
4. Take the inverse transform of the result: $\mathcal{F}^{-1}(\mathcal{F}(M) \cdot \mathcal{F}(S'))$.

Put simply, the convolution theorem states:

$$M * S = \mathcal{F}^{-1}(\mathcal{F}(M) \cdot \mathcal{F}(S'))$$

or equivalently that

$$\mathcal{F}(M * S) = \mathcal{F}(M) \cdot \mathcal{F}(S').$$

Although this might seem like an unnecessarily clumsy and roundabout way of computing something so simple as a convolution, it can have enormous speed advantages if S is large.

For example, suppose we wish to convolve a 512 × 512 image with a 32 × 32 filter. To do this directly would require $32^2 = 1024$ multiplications for each pixel, of which there are 512 × 512 = 262,144. Thus there will be a total of 1024 × 262,144 = 268,435,456 multiplications needed. Now look at applying the DFT (using an FFT algorithm). Each row requires 4608 multiplications by Table 7.1; there are 512 rows, so a total of 4608 × 512 = 2,359,296 multiplications; the same must be done again for the columns. Thus to obtain the DFT of the image requires 4,718,592 multiplications. We need the same amount to obtain the DFT of the filter, and for the inverse DFT. We also require 512 × 512 multiplications to perform the product of the two transforms.

Thus, the total number of multiplications needed to perform convolution using the DFT is

$$4{,}718{,}592 \times 3 + 262{,}144 = 14{,}417{,}920$$

which is an enormous saving compared to the direct method.

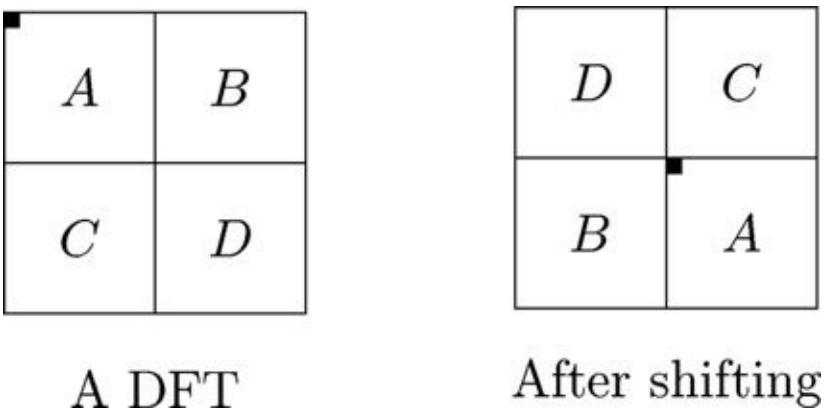
The DC coefficient. The value $F(0, 0)$ of the DFT is called the *DC coefficient*. If we put $u = v = 0$ in the definition given in Equation 7.4, then

$$F(0,0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \exp(0) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y).$$

That is, this term is equal to the sum of *all* terms in the original matrix.

Shifting. For purposes of display, it is convenient to have the DC coefficient in the center of the matrix. This will happen if all elements $f(x, y)$ in the matrix are multiplied by $(-1)^{x+y}$ before the transform. Figure 7.8 demonstrates how the matrix is shifted by this method. In each diagram, the DC coefficient is the top left-hand element of submatrix A, and is shown as a black square.

Figure 7.8: Shifting a DFT



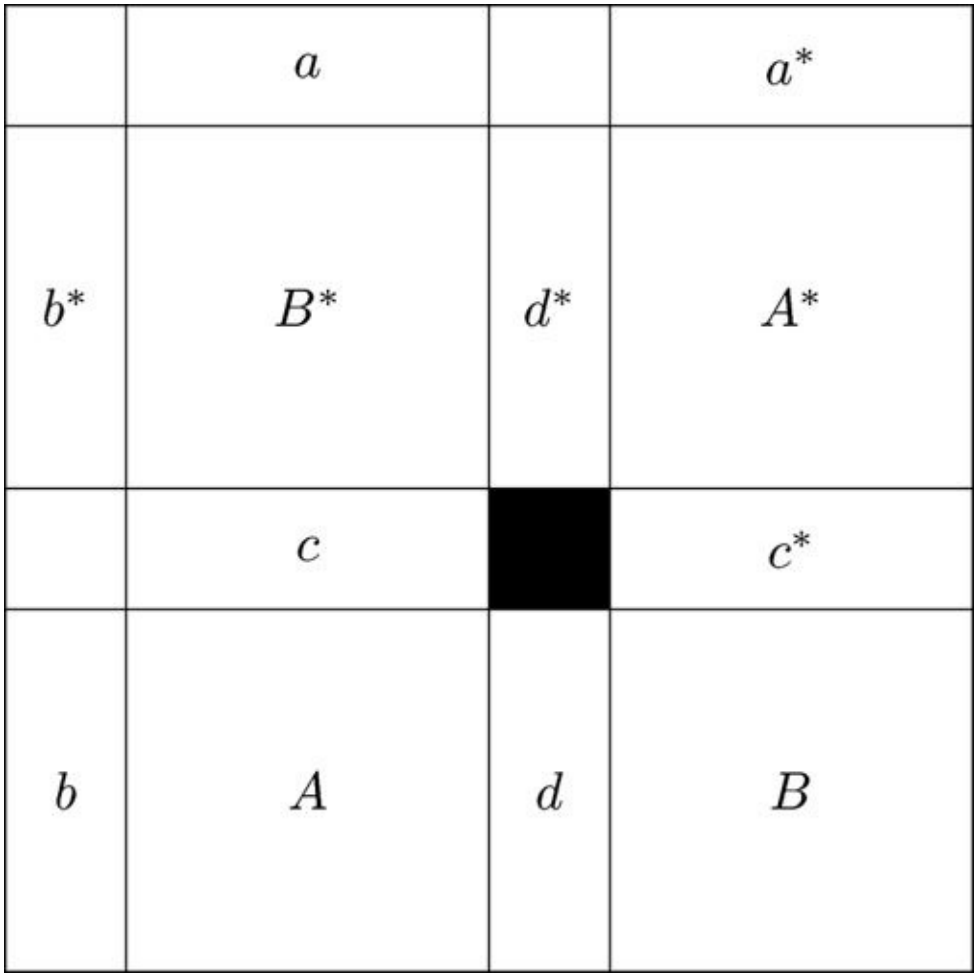
Conjugate symmetry An analysis of the Fourier Transform definition leads to a symmetry property; if we make the substitutions $u = -u$ and $v = -v$ in Equation 7.4, then

$$\mathcal{F}(u, v) = \mathcal{F}^*(-u + pM, -v + qN)$$

for any integers p and q. This means that half of the transform is a mirror image of the conjugate of the other half. We can think of the top and bottom halves, or the left and right halves, being mirror images of the conjugates of each other.

Figure 7.9 demonstrates this symmetry in a shifted DFT. As with Figure 7.8, the black square shows the position of the DC coefficient. The symmetry means that its information is given in just half of a transform, and the other half is redundant.

Figure 7.9: Conjugate symmetry in the DFT



Displaying transforms. Having obtained the Fourier Transform $F(u, v)$ of an image $f(x, y)$, we would like to see what it looks like. As the elements $F(u, v)$ are complex numbers, we can't view them directly, but we can view their magnitude $|F(u, v)|$. Since these will be numbers of type `double`, generally with large range, we would need to scale these absolute values so that they can be displayed.

One trouble is that the DC coefficient is generally very much larger than all other values. This has the effect of showing a transform as a single white dot surrounded by black. One way of stretching out the values is to take the logarithm of $|F(u, v)|$ and to display

$$\log(1 + |F(u, v)|).$$

The display of the magnitude of a Fourier Transform is called the *spectrum* of the transform. We shall see some examples later.

7.6 Experimenting with Fourier Transforms

The relevant functions for us are:

- `fft`, which takes the DFT of a vector
- `ifft`, which takes the inverse DFT of a vector
- `fft2`, which takes the DFT of a matrix
- `ifft2`, which takes the inverse DFT of a matrix
- `fftshift`, which shifts a transform as shown in Figure 7.8

of which we have seen the first two above. These functions are available in MATLAB and Octave, and can be brought into the top namespace of Python with the command

```
from numpy.fft import *
```

Before attacking a few images, let's take the Fourier Transform of a few small matrices to get more of an idea what the DFT "does."

Example 1. Suppose we take a constant matrix $f(x, y) = 1$. Going back to the idea of a sum of corrugations, then *no* corrugations are required to form a constant. Thus, we would hope that the DFT consists of a DC coefficient and zeros everywhere else. We will use the `ones` function, which produces an n x n matrix consisting of 1's, where n is an input to the function.

```
>> a1 = ones(8);
>> fft2(a1)
```

MATLAB/Octave

The result is indeed as we expected:

```
ans =
      64     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
```

MATLAB/Octave

Note that the DC coefficient is indeed the sum of all the matrix values.

Example 2. Now we will take a matrix consisting of a single corrugation:

```
>> a2 = [100 200; 100 200];
>> a2 = repmat(a2,4,4)

ans =
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200
     100   200   100   200   100   200   100   200

>> af2 = fft2(a2)

ans =

    9600     0     0     0 -3200     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
```

MATLAB/Octave

What we have here is really the sum of two matrices: a constant matrix each element of which is 150, and a corrugation which alternates −50 and 50 from left to right. The constant matrix alone would produce (as in example 1) a DC coefficient alone of value 64 × 150 = 9600; the corrugation a single value. By linearity, the DFT will consist of just the two values.

Example 3. We will take here a single step edge:

```
>> a3 = [zeros(8,4) ones(8,4)]
  a3 =

     0     0     0     0     1     1     1     1
     0     0     0     0     1     1     1     1
     0     0     0     0     1     1     1     1
     0     0     0     0     1     1     1     1
     0     0     0     0     1     1     1     1
     0     0     0     0     1     1     1     1
     0     0     0     0     1     1     1     1
     0     0     0     0     1     1     1     1
```

MATLAB/Octave

Now we shall perform the Fourier Transform with a shift, to place the DC coefficient in the center, and since it contains some complex values, for simplicity we shall just show the rounded absolute values:

```
>> af3 = fftshift(fft2(a3));
>> round(abs(af3))

ans =

       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     9     0    21    32    21     0     9
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
       0     0     0     0     0     0     0     0
```

MATLAB/Octave

The DC coefficient is of course the sum of all values of `a`; the other values may be considered to be the coefficients of the necessary sine functions required to from an edge, as given in Equation 7.1. The mirroring of values about the DC coefficient is a consequence of the symmetry of the DFT.

All these can also be easily performed in Python, with the three images constructed as

```
In:  a1 = ones((8,8))
In:  a2 = np.array([[100, 200],[100,200]])
In:  a2 = np.tile(a2,(4,4))
In:  a3 = np.hstack([zeros((8,4)),ones((8,4))])
```

Python

Then the Fourier spectra can be obtained by very similar commands to those above, but converting to integers for easy viewing:

```
In:  int16(fft2(a1))
In:  int16(fft2(a1))
In:  af3 = fftshift(fft2(a3))
In:  int16(np.round(abs(af3),0))
```

Python

7.7 Fourier Transforms of Synthetic Images

Before experimenting with images, it will be helpful to have a program that will enable the viewing of Fourier spectra, either with absolute values, or with log-scaling. Such a program, called `fftshow`, is given in all our languages at the end of the chapter.

Now we can create a few simple images, and see what the Fourier Transform produces.

Example 1. We shall produce a simple image consisting of a single edge, first in MATLAB/Octave

```
>> a = [zeros(256,128) ones(256,128)];
>> af = fftshift(fft2(a));
```

MATLAB/Octave

The image is displayed on the left in Figure 7.10. The spectrum can be displayed either directly or with log-scaling:

```
>> afl = abs(af);
>> imshow(mat2gray(afl))
>> afl2 = log(1+abs(af));
>> imshow(mat2gray(afl2));
```

MATLAB/Octave

Alternatively, the largest value can be isolated: this will be the DC coefficient that, because of shifting, will be at position (129, 129), and divide the spectrum by this value for display. Either of the first two commands will work:

```
>> mx = max(afl(:))
>> mx = afl(129,129)
>> imshow(afl/mx)
```

MATLAB/Octave

The image and the two displays are shown in Figure 7.10.

In Python, the commands for constructing the image and viewing its Fourier spectrum are:

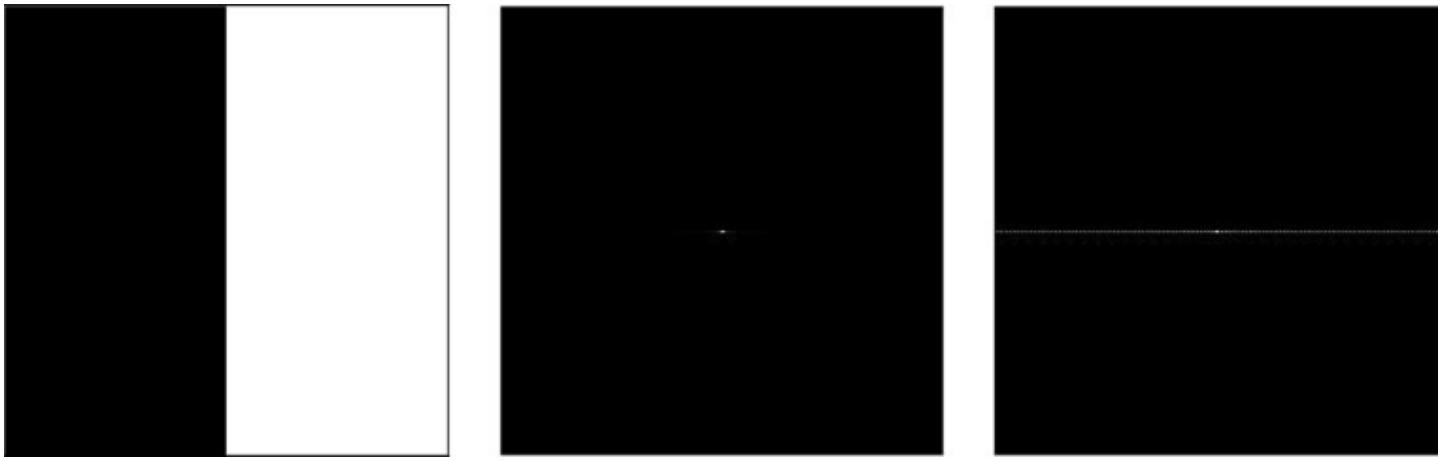
```
In:  a = np.hstack([zeros((256,128)),ones((256,128))])
In:  af = fftshift(fft2(a))
In:  afl = abs(af)
In:
```

Python

The image and its Fourier spectrum shown first as absolute values and then as log-scaled are shown in Figure 7.10. We observe immediately that the final (right-most) result is similar (although larger) to Example 3 in the previous section.

Example 2. Now we will create a box, and then its Fourier Transform:

Figure 7.10: A single edge and its DFT



```
>> a = zeros(256,256);
>> a(78:178,78:178) = 1;
>> imshow(a)
>> af = fftshift(fft2(a));
>> figure, imshow(mat2gray(log(1+abs(af))))
```

MATLAB/Octave

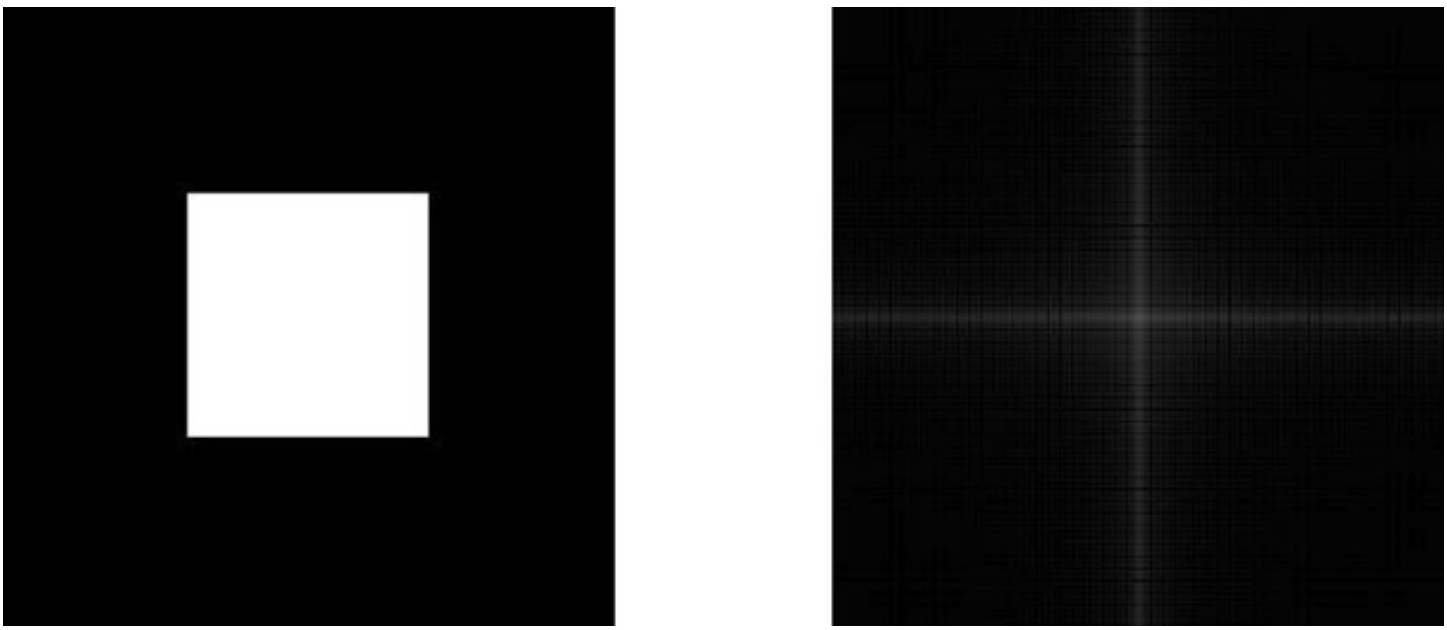
In Python, we can use the `rescale_intensity` function to scale the output to fit a given range:

```
In:  a = zeros((256,256))
In:  a[77:177,77:177] = 1
In:  af = fftshift(fft2(a))
In:  afl = ex.rescale_intensity(np.log(1+abs(af)),outrange=(0.0,1.0))
In:  io.imshow(afl)
```

Python

The box is shown on the left in Figure 7.11, and its Fourier transform is shown on the right.

Figure 7.11: A box and its DFT



Example 3. Now we shall look at a box rotated 45°, first in MATLAB/Octave.

```
>> [x,y] = meshgrid(1:256,1:256);
>> b = (x+y<329)&(x+y>182)&(x-y>-67)&(x-y<73);
>> imshow(b)
>> bf = fftshift(fft2(b));
>> figure, imshow(mat2gray(log(1+abs(af))))
```

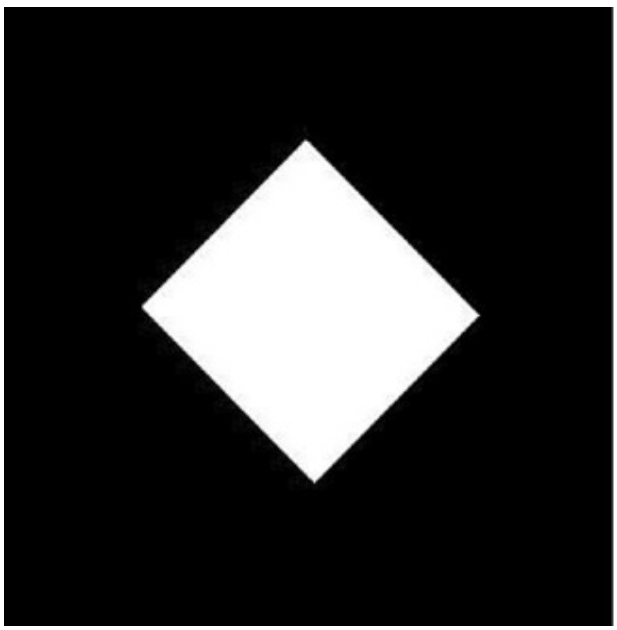
MATLAB/Octave

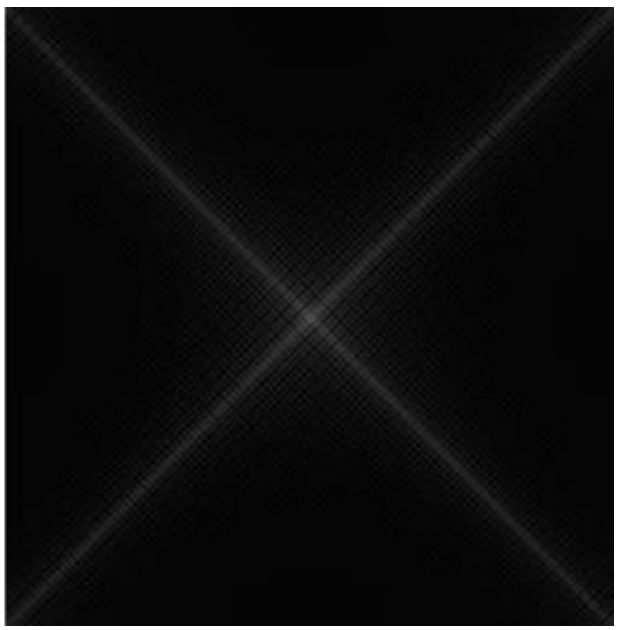
and next in Python:

```
In:  x,y = meshgrid(range(256),range(256))
In:  b = (x+y<329)&(x+y>182)&(x-y>-67)&(x-y<73);
In:  io.imshow(b)
In:  bf = fftshift(fft2(b))
In:  bfl = log(1+abs(bf))
In:  io.imshow(ex.rescale_intensity(bfl,out_range=(0.0,1.0)))
```

Python

The results are shown in Figure 7.12. Note that the transform of the rotated box is the





rotated transform of the original box.

Figure 7.12: A rotated box and its DFT

```
>> [x,y] = meshgrid(-128:217,-128:127);
>> z = sqrt(x.^2+y.^2);
>> c = (z<15);
```

MATLAB/Octave

Example 4. We will create a small circle, and then transform it:

```
In:  ar = np.arange(-128,128)
In:  x,y = meshgrid(ar,ar)
In:  z = sqrt(x**2+y**2)
In:  c = (z<15)*1
```

Python

or

```
>> cf = fftshift(fft2(c));
>> imshow(mat2gray(log(1+abs(cf))))
```

MATLAB/Octave

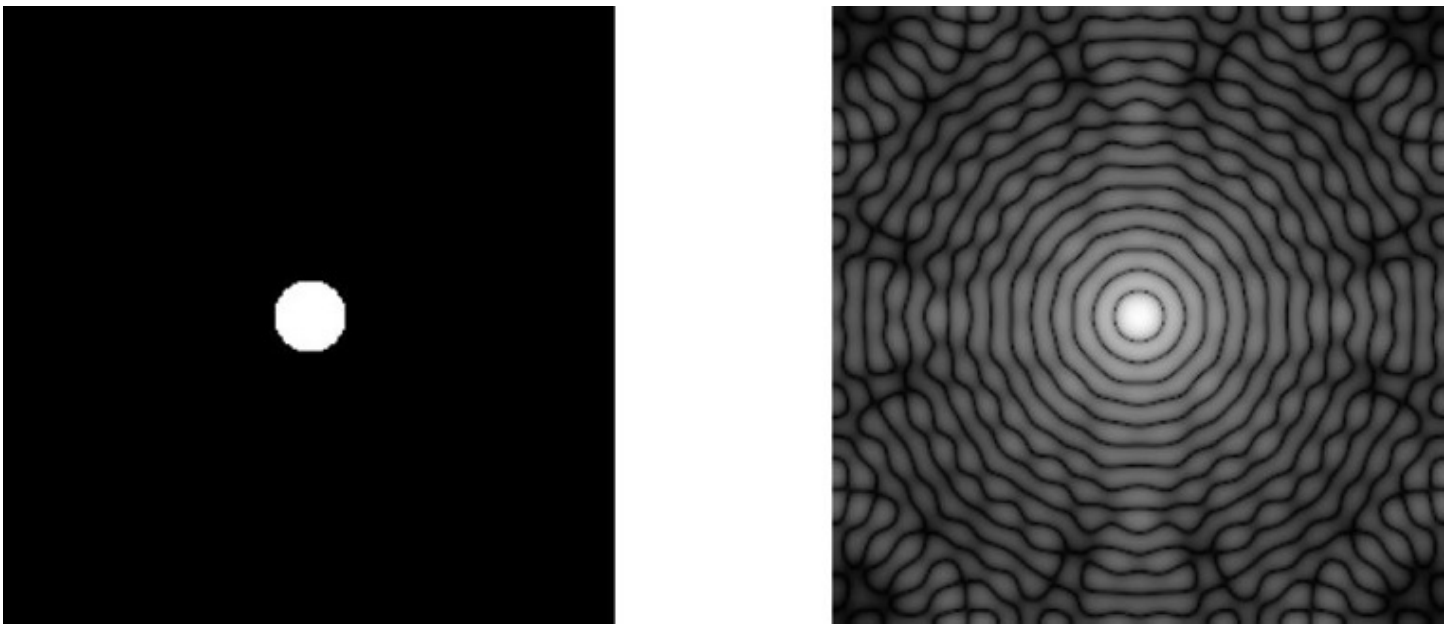
The result is shown on the left in Figure 7.13. Now we will create its Fourier Transform and display it:

```
In:  cf = fftshift(fft2(c))
In:  cfl = log(1+abs(cf))
In:  io.imshow(ex.rescale_intensity(cfl,out_range=(0.0,1.0)))
```

Python

and this is shown on the right in Figure 7.13. Note the "ringing" in the Fourier transform.

Figure 7.13: A circle and its DFT



This is an artifact associated with the sharp cutoff of the circle. As we have seen from both the edge and box images in the previous examples, an edge appears in the transform as a line of values at right angles to the edge. We may consider the values on the line as being the coefficients of the appropriate corrugation functions which sum to the edge. With the circle, we have lines of values radiating out from the circle; these values appear as circles in the transform.

A circle with a gentle cutoff, so that its edge appears blurred, will have a transform with no ringing. Such a circle can be made with the commands (given z above):

```
>> c2 = 1./(1+(z./15).^2);
```

MATLAB/Octave

or

```
In:  c2 = 1/(1+(z/15)**2)
```

Python

This image appears as a blurred circle, and its transform is very similar—check them out!

7.8 Filtering in the Frequency Domain

We have seen in Section 7.5 that one of the reasons for the use of the Fourier Transform in image processing is due to the convolution theorem: a spatial convolution can be performed by element-wise multiplication of the Fourier Transform by a suitable "filter matrix." In this section, we shall explore some filtering by this method.

Ideal Filtering

Low Pass Filtering

Suppose we have a Fourier Transform matrix F, shifted so that the DC coefficient is in the center. Recall from Figure 7.6 that the low frequency components are toward the center. This means we can perform low pass filtering by multiplying the transform by a matrix in such a way that center values are maintained, and values away from the center are either removed or minimized. One way to do this is to multiply by an *ideal low pass matrix*, which is a binary matrix m defined by:

$$m(x,y) = \begin{cases} 1 & \text{if } (x,y) \text{ is closer to the center than some value } D, \\ 0 & \text{if } (x,y) \text{ is further from the center than } D. \end{cases}$$

The circle c displayed in Figure 7.13 is just such a matrix, with D = 15. Then the inverse Fourier Transform of the element-wise product of F and m is the result we require:

$$\mathcal{F}^{-1}(F \cdot m).$$

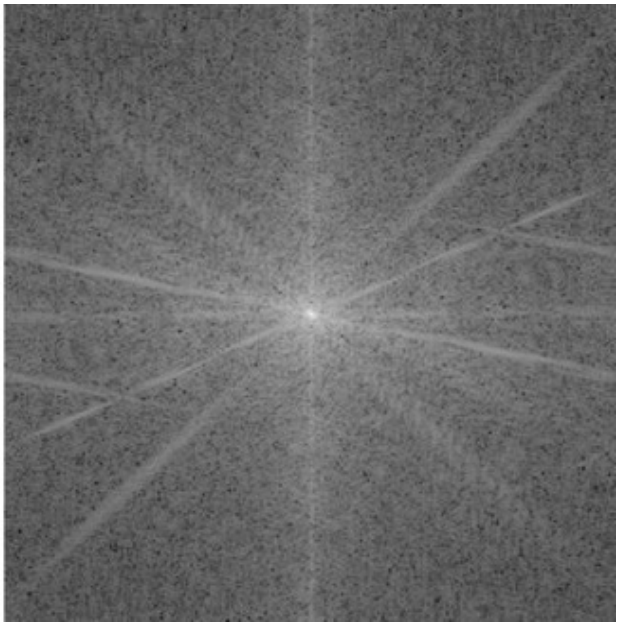
Let's see what happens if we apply this filter to an image. First we obtain an image and its DFT.

```
>> cm = imread('cameraman.png');
>> cf = fftshift(fft2(cm));
>> imshow(mat2gray(log(1+abs(cf))))
```

MATLAB/Octave

The cameraman image and its DFT are shown in Figure 7.14. Now we can perform a low

Figure 7.14: The “cameraman” image and its DFT



pass filter by multiplying the transform matrix by the circle matrix we created earlier (recall that “dot asterisk” is the MATLAB syntax for element-wise multiplication of two matrices):

```
>> cfl = cf.*c;
>> imshow(mat2gray(log(1+abs(cfl))))
```

MATLAB/Octave

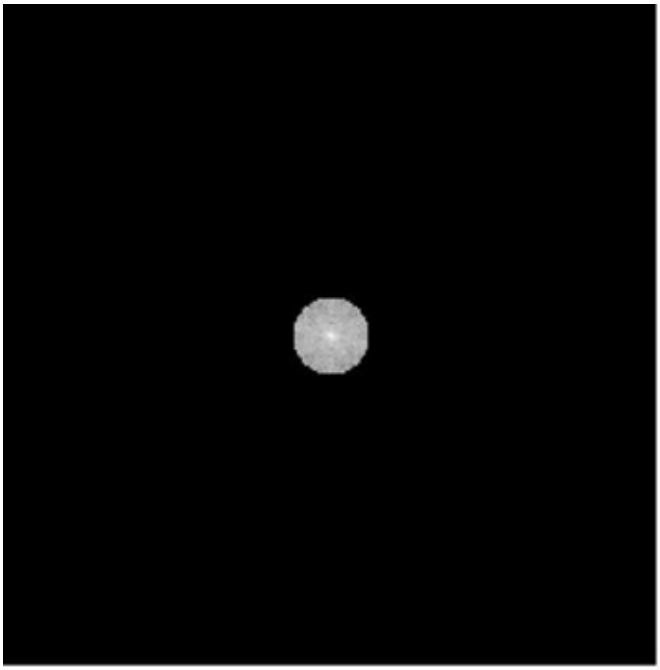
and this is shown in Figure 7.15(a). Now we can take the inverse transform and display the result:

```
>> cfli = ifft2(cfl);
>> imshow(mat2gray(abs(cfli)))
```

MATLAB/Octave

and this is shown in Figure 7.15(b). Note that even though `cfli` is supposedly a matrix of real numbers, we are still using `abs` to obtain displayable values. This is because the `fft2` and `fft2` functions, being numeric, will not produce mathematically perfect results, but rather very close numeric approximations. So, taking the absolute values of the result rounds out any errors obtained during the transform and its inverse. Note the “ringing” about the edges in this image. This is a direct result of the sharp cutoff of the circle. The ringing as shown in Figure 7.13 is transferred to the image.

Figure 7.15: Applying ideal low pass filtering

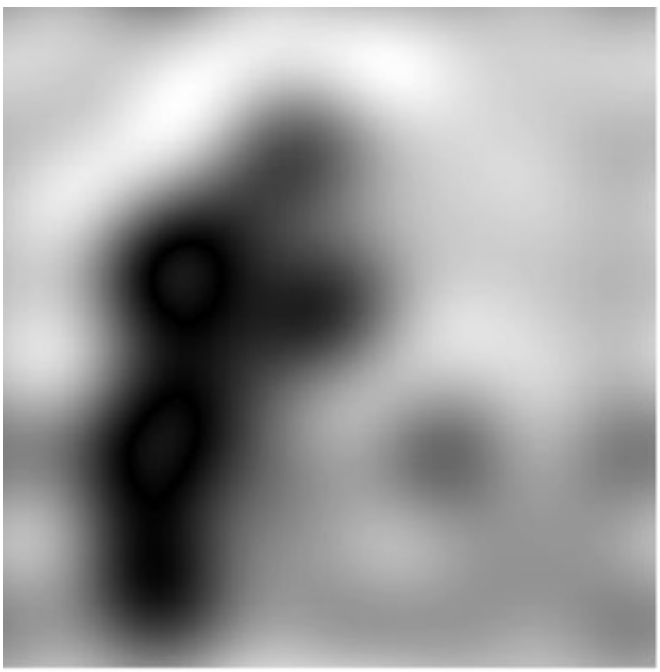


(a) Ideal filtering on the DFT

(b) After inversion

We would expect that the smaller the circle, the more blurred the image, and the larger the circle; the less blurred. Figure 7.16 demonstrates this, using cutoffs of 5 and 30. Notice that ringing is still present, and clearly visible in Figure 7.16(b).

Figure 7.16: Ideal low pass filtering with different cutoffs



(a) Cutoff of 5

(b) Cutoff of 30

All these operations can be performed with almost the same commands in Python; first reading the image and producing its Fourier Transform:

```
In: cm = io.imread('cameraman.png')
In: cf = fftshift(fft2(cm))
```

Python

and then the low pass filter with a circle of cutoff 15:

```
In: d = 15
In: ar = range(-128,128)
In: x,y = meshgrid(ar,ar)
In: c = (x**2+y**2<d**2)*1
```

Python

Finally, the filter can be applied to the Fourier Transform and the inverse obtained and viewed:

```
In: cfl = cf*c
In: io.imshow(abs(ifft2(cfl)))
```

Python

This will produce the result shown in Figure 7.15. The other images can be produced by varying the value of *d* in the definition of the ideal filter.

High Pass Filtering

Just as we can perform low pass filtering by keeping the center values of the DFT and eliminating the others, so high pass filtering can be performed by the opposite: eliminating center values and keeping the others. This can be done with a minor modification of the preceding method of low pass filtering. First we create the circle:

```
>> [x,y] = meshgrid(-128:127,-128:127);
>> z = sqrt(x.^2+y.^2);
>> c = z>15;
```

MATLAB/Octave

and then multiply it by the DFT of the image:

```
>> cfh = cf.*c;
>> imshow(mat2gray(log(1+abs(cfh))))
```

MATLAB/Octave

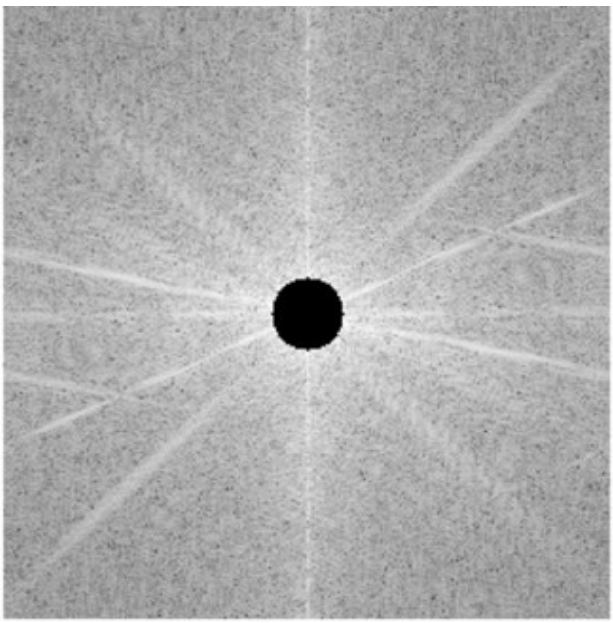
This is shown in Figure 7.17(a). The inverse DFT can be easily produced and displayed:

```
>> cfhi = ifft2(cfh);
>> figure,imshow(abs(cfhi))
```

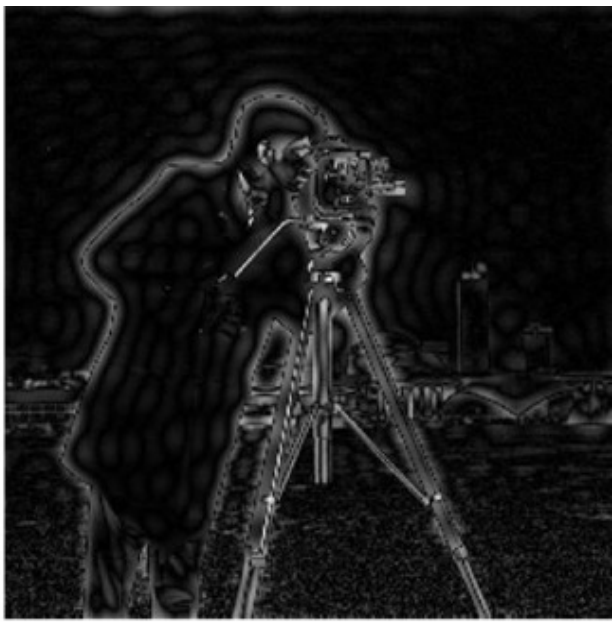
MATLAB/Octave

and this is shown in Figure 7.17(b). As with low pass filtering, the size of the circle influences the information available to the inverse DFT, and hence the final result. Figure 7.18 shows some results of ideal high pass filtering with different cutoffs. If the cutoff is large, then more information is removed from the transform, leaving only the highest frequencies. This can be observed in Figure 7.18(c) and (d); only the edges of the image remain. If we have small cutoff, such as in Figure 7.18(a), we are only removing a small amount of the transform.

Figure 7.17: Applying an ideal high pass filter to an image

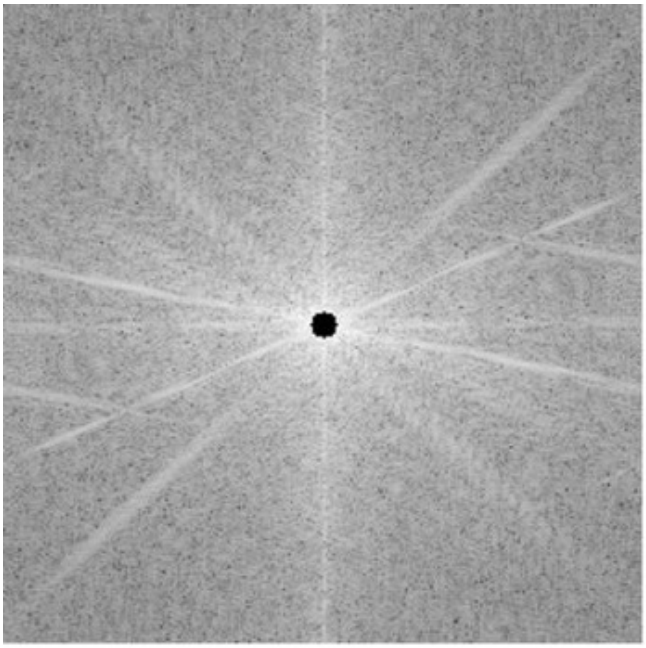


(a) The DFT after high pass filtering



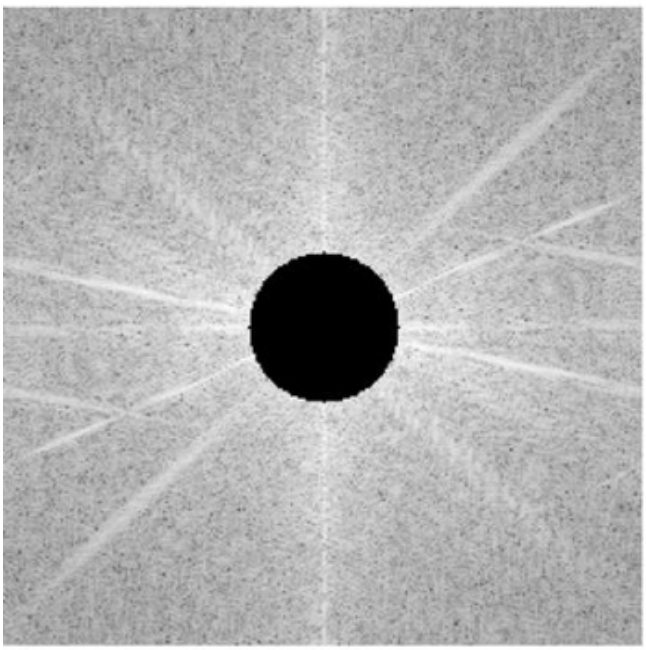
(b) The resulting image

Figure 7.18: Ideal high pass filtering with different cutoffs



(a) Cutoff of 5

(b) The resulting image



(a) Cutoff of 30

(b) The resulting image

We would thus expect that only the lowest frequencies of the image would be removed. And this is indeed true, as seen in Figure 7.18(b); there is some grayscale detail in the final image, but large areas of low frequency are close to zero.

In Python, the commands are almost the same as before, except that with a cutoff of d the ideal filter will be produced with

```
In: c = (x**2+y**2>d**2)*1
```

Python

Butterworth Filtering

Ideal filtering simply cuts off the Fourier Transform at some distance from the center. This is very easy to implement, as we have seen, but has the disadvantage of introducing unwanted artifacts: ringing, into the result. One way of avoiding this is to use as a filter matrix a circle with a less sharp cutoff. A popular choice is to use *Butterworth filters*.

Before we describe these filters, we shall look again at the ideal filters. As these are radially symmetric about the center of the transform, they can be simply described in terms of their cross-sections. That is, we can describe the filter as a function of the distance x from the center. For an ideal low pass filter, this function can be expressed as

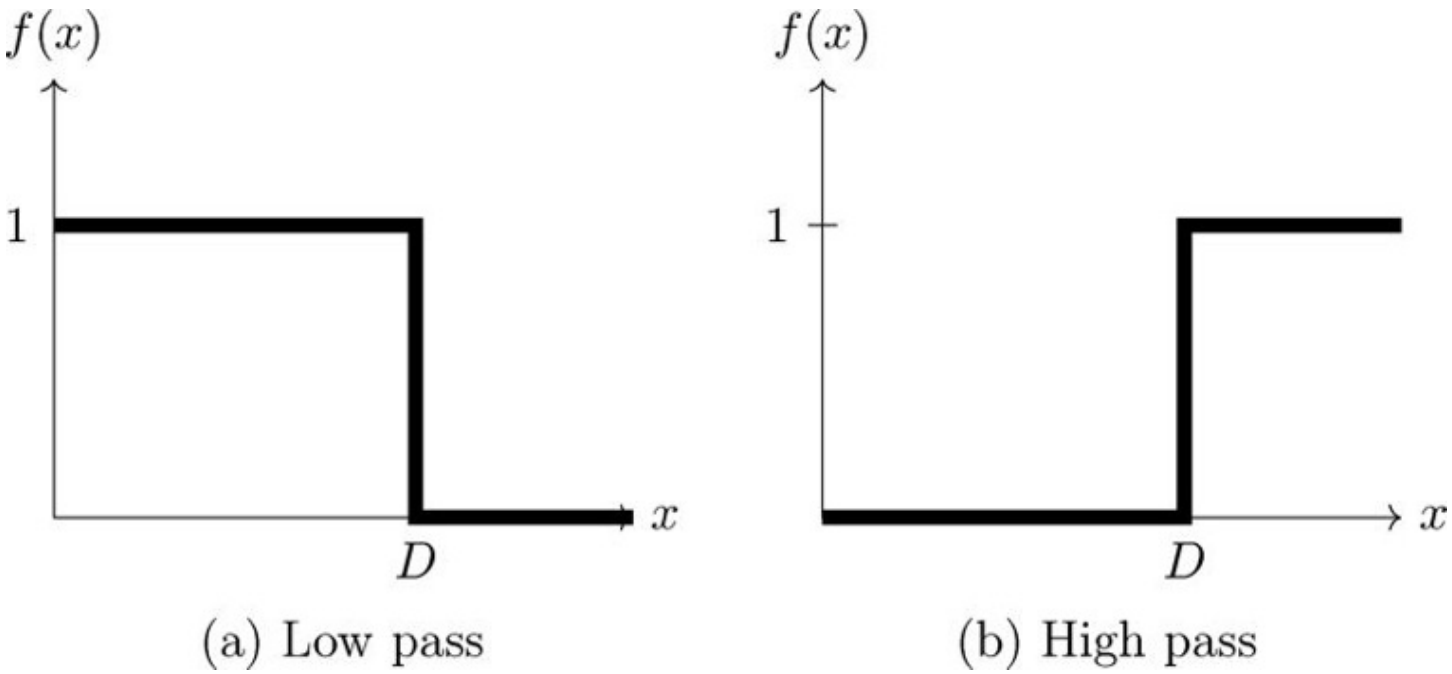
$$f(x) = \begin{cases} 1 & \text{if } x < D, \\ 0 & \text{if } x \geq D \end{cases}$$

where D is the cutoff radius. Then the ideal high pass filters can be described similarly:

$$f(x) = \begin{cases} 1 & \text{if } x > D, \\ 0 & \text{if } x \leq D \end{cases}$$

These functions are illustrated in Figure 7.19. Butterworth filter functions are based on the

Figure 7.19: Ideal filter functions



following functions for low pass filters:

$$f(x) = \frac{1}{1 + (x/D)^{2n}}$$

and for high pass filters:

$$f(x) = \frac{1}{1 + (D/x)^{2n}}$$

where in each case the parameter n is called the *order* of the filter. The size of n dictates the sharpness of the cutoff. These functions are illustrated in figures 7.20 and 7.21.

Note that on a two-dimensional grid, where x in the above formulas may be replaced with $\sqrt{x^2+y^2}$ as distance from the grid's center, assuming the grid to be centered at (0, 0), then the formulas for the low pass and high pass filters may be written

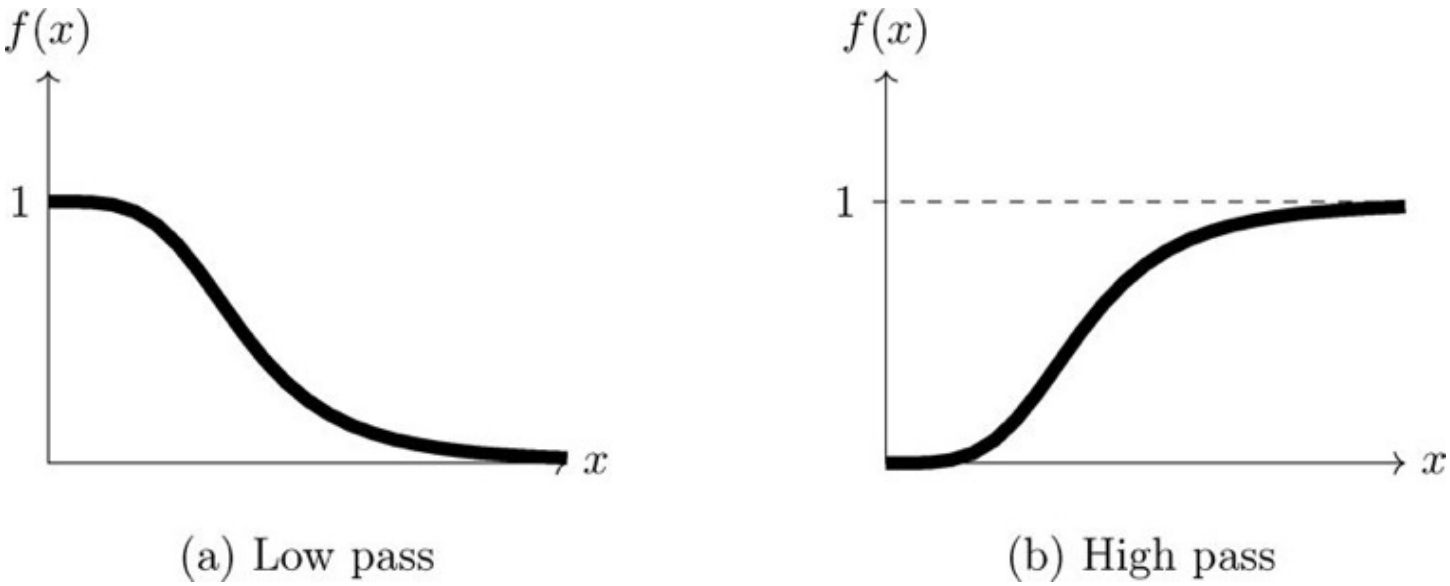
$$f(x,y) = \frac{1}{1+\left(\frac{x^2+y^2}{D^2}\right)^n}$$

and

$$f(x,y) = \frac{1}{1+\left(\frac{D^2}{x^2+y^2}\right)^n}$$

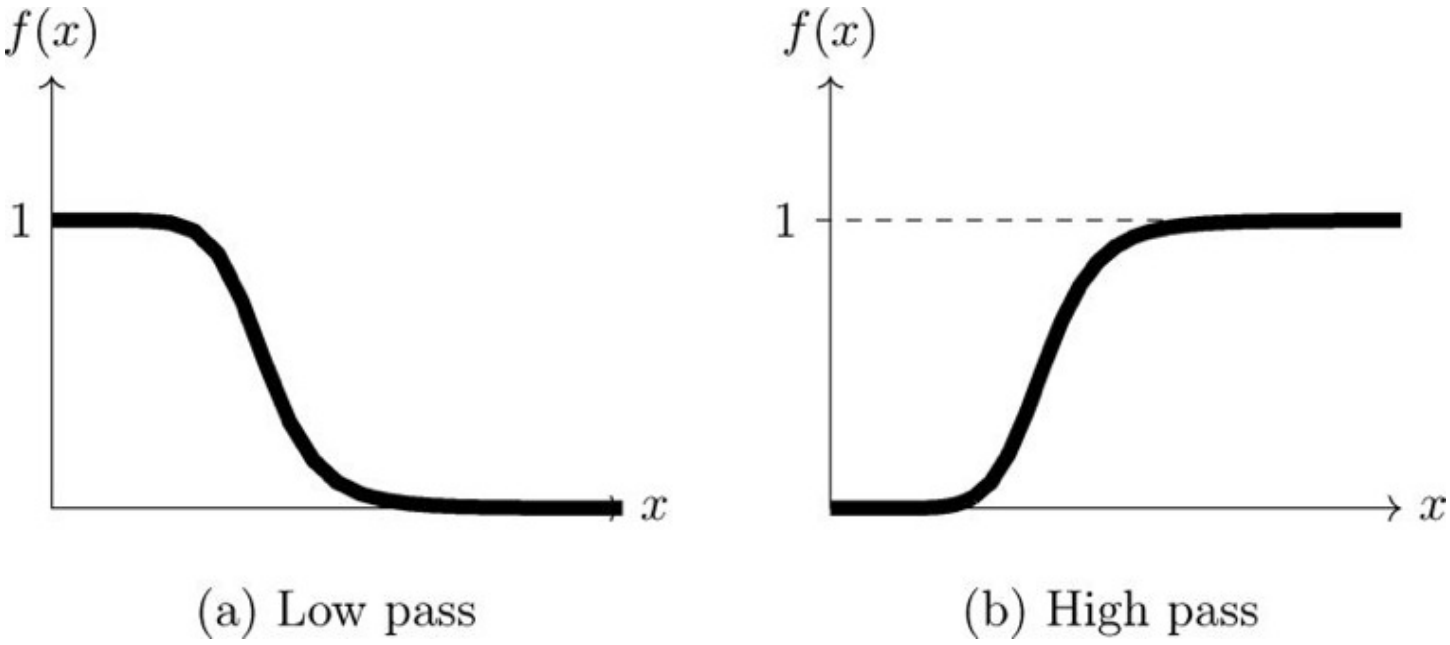
respectively.

Figure 7.20: Butterworth filter functions with n = 2



(a) Low pass (b) High pass

Figure 7.21: Butterworth filter functions with $n = 4$



It is easy to implement these filters; here are the commands to produce a Butterworth low pass filter of size 256 × 256 with $D = 15$ and order $n = 2$:

```
>> [x,y] = meshgrid(-128:217,-128:127));
>> bl = 1./(1+((x.^2+y.^2)/15).^2);
```

MATLAB/Octave

and in Python:

```
In:  ar = arange(-128,128,1.0)
In:  x,y = meshgrid(ar,ar)
In:  bl = 1/(1+((x**2+y**2)/15**2)**2)
```

Python

Note that in Python it is essential that the elements of the array have floating point values, so that all arithmetic will be done as floating point, rather than integers. Hence, we start by making the indexing array consist of floats by using `arange` instead of `range`.

Since a Butterworth high pass filter can be obtained by subtracting a low pass filter from 1, the above commands can be used to create high pass filters.

To apply a Butterworth low pass filter in Python, then, with $n = 2$ and $D = 15.0$, we can use the commands above, and then:

```
>> ar = range(-128,127)
>> x,y = meshgrid(ar,ar)
>> D = 15.0
>> bl = 1.0/(1.0+((x**2+y**2)/D**2)**2)
>> cfbl = cf*bl
>> io.imshow(ex.rescale_intensity(abs(ifft2(cfbl)),out_range=(0.0,1.0)))
```

Python

And to apply a Butterworth low pass filter to the DFT of the cameraman image in MATLAB/Octave, create the filter array as above and then apply it to the image transform:

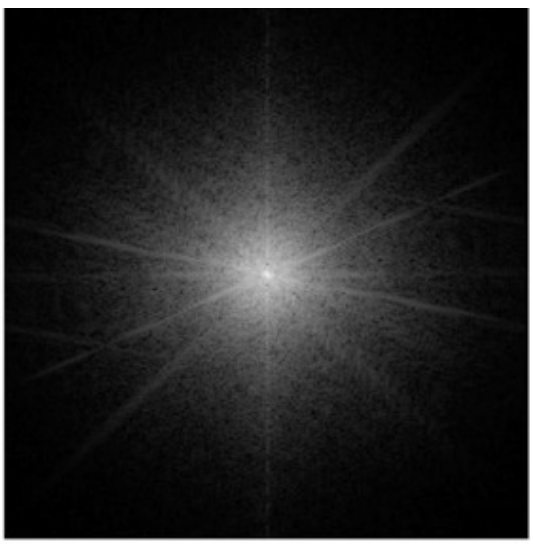
```
>> cfbl = cf.*bl;
>> figure,fftshow(cfbl,'log')
```

MATLAB/Octave

and this is shown in Figure 7.22(a). Note that there is no sharp cutoff as seen in Figure 7.15; also that the outer parts of the transform are not equal to zero, although they are dimmed considerably. Performing the inverse transform and displaying it as we have done previously produces Figure 7.22(b). This is certainly a blurred image, but the ringing seen in Figure 7.15 is completely absent. Compare the transform after multiplying with a Butterworth

filter (Figure 7.22(a)) with the original transform (in Figure 7.14). The Butterworth filter does cause an attenuation of values away from the center, even if they don't become suddenly zero, as with the ideal low pass filter in Figure 7.15.

Figure 7.22: Butterworth low pass filtering



(a) The DFT after Butterworth low pass filtering

(b) The resulting image

We can apply a Butterworth high pass filter similarly, first by creating the filter and applying it to the image transform:

```
In: bh = 1- 1/(1+((x**2+y**2)/D**2)**2)
In: cfbh = cf*bh;
```

Python

and then inverting and displaying the result:

```
In: io.imshow(abs(ifft2(cfbh)))
```

Python

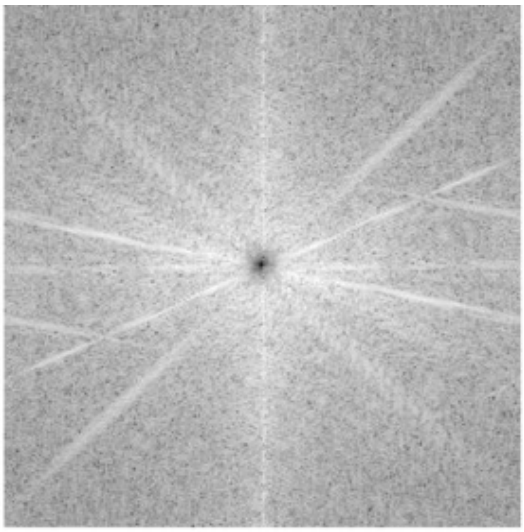
In MATLAB/Octave the commands are similar:

```
>> D = 15
>> bh = 1 - 1./(1+((x.^2+y.^2)/15).^2);
>> cfbh = cf.*bh;
>> imshow(mat2gray(log(1+abs(cfbh))))
>> figure,imshow(mat2gray(abs(ifft2(cfbh))))
```

MATLAB/Octave

The images are shown in Figure 7.23

Figure 7.23: Butterworth high pass filtering



(a) The DFT after Butterworth high pass filtering

(b) The resulting image

Gaussian Filtering

We have met Gaussian filters in Chapter 5, and we saw that they could be used for low pass filtering. However, we can also use Gaussian filters in the frequency domain. As with ideal and Butterworth filters, the implementation is very simple: create a Gaussian filter, multiply it by the image transform, and invert the result. Since Gaussian filters have the very nice mathematical property that a Fourier Transform of a Gaussian is a Gaussian, we should get exactly the same results as when using a linear Gaussian spatial filter.

Gaussian filters may be considered to be the most “smooth” of all the filters we have discussed so far, with ideal filters the least smooth and Butterworth filters in the middle.

In Python, Gaussian filters are provided in the `ndimage` of `scipy`, and there is a wrapper in `skimage.filter`. However, we can also create a filter for ourselves. Using the cameraman image, and a standard deviation $\sigma = 10$, define:

```
In: sigma = 10
In: g = exp(-(x**2+y**2)/sigma**2)
In: g1 = g1/g1.sum()
```

Python

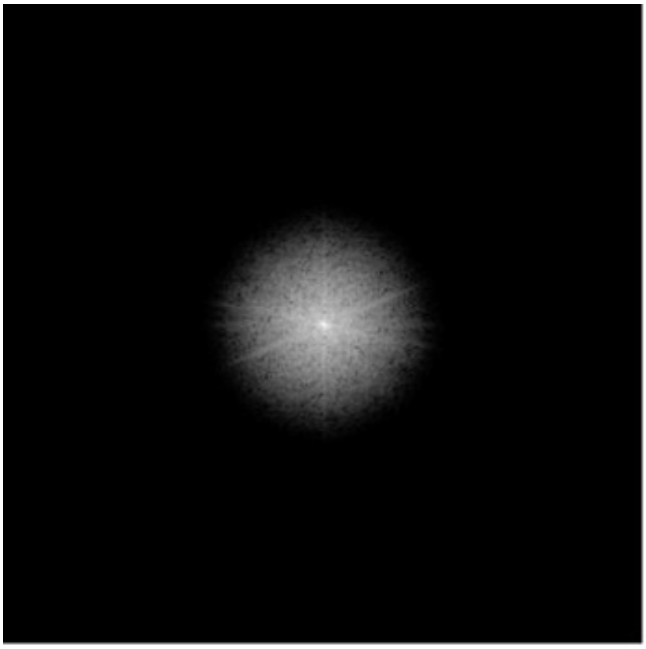
where the last command is to ensure that the sum of all filter elements is one. This filter can then be applied to the image in the same way as the ideal and Butterworth filters above:

```
In: cg1 = cf*g1
In: io.imshow(abs(ifft2(cg1)))
```

Python

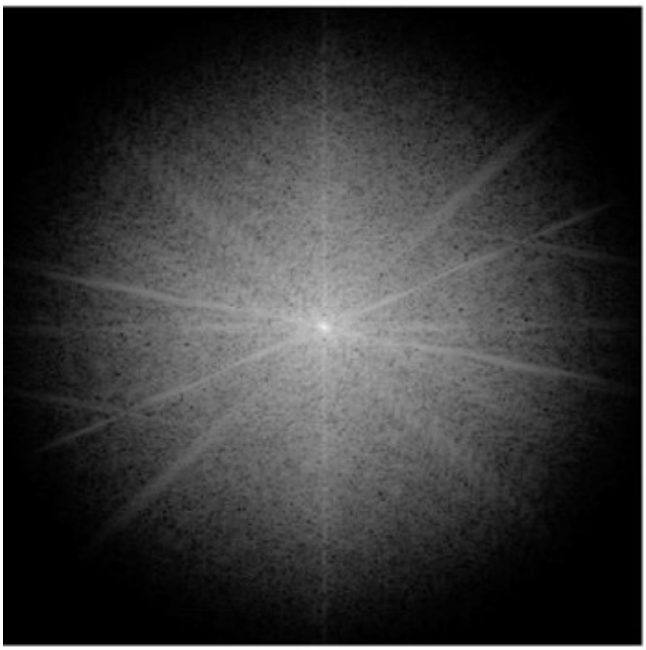
and the results are shown in Figure 7.24(b) and (d). A high pass filter can be defined by

Figure 7.24: Applying a Gaussian low pass filter in the frequency domain



(a) $\sigma = 10$

(b) Resulting image



(c) $\sigma = 30$

(d) Resulting image

first scaling a low pass filter so that its maximum value is one, and then subtracting it from 1:

```
In: h1 = 1-g/g.max()
```

Python

and it can then be applied to the image:

```
In: ch1 = cf*h1
In: io.imshow(abs(ifft2(ch1)))
```

Python

The same sequence of commands can be produced starting with σ = 30, and the images are shown in Figure 7.25.

Figure 7.25: Applying a Gaussian high pass filter in the frequency domain

(a) Using $\sigma = 10$

(b) Using $\sigma = 30$

With MATLAB or Octave, Gaussian filters can be created using the `fspecial` function, and applied to our transform.

```
>> g1=mat2gray(fspecial('gaussian',256,10));
>> cg1=cf.*g1;
>> imshow(mat2gray(log(1+abs(cg1))))
>> g2=mat2gray(fspecial('gaussian',256,30));
>> cg2=cf.*g2;
>> imshow(mat2gray(log(1+abs(cg1))))
```

MATLAB/Octave

Note the use of the `mat2gray` function. The `fspecial` function on its own produces a low pass Gaussian filter with a very small maximum:

```
>> g=fspecial('gaussian',256,10);
>> format long, max(g(:)), format

ans =

   0.00158757552679
```

MATLAB/Octave

The reason is that `fspecial` adjusts its output to keep the volume under the Gaussian function always 1. This means that a wider function, with a large standard deviation, will have a low maximum. So we need to scale the result so that the central value will be 1; and `mat2gray` does that automatically.

The transforms are shown in Figure 7.24(a) and (c). In each case, the final parameter of the `fspecial` function is the standard deviation; it controls the width of the filter. Clearly, the larger the standard deviation, the wider the function, and so the greater amount of the transform is preserved.

The results of the transform on the original image can be produced using the usual sequence of commands:

```
>> cgi1 = ifft2(cg1);
>> cgi2 = ifft2(cg2);
>> fftshow(cgi1,'abs');
>> fftshow(cgi2,'abs');
```

MATLAB/Octave

We can apply a high pass Gaussian filter easily; we create a high pass filter by subtracting a low pass filter from 1.

```
>> h1 = 1-g1;
>> h2 = 1-g2;
>> ch1 = cf.*h1;
>> ch2 = cf.*h2;
>> chi1 = ifft2(ch1);
>> chi2 = ifft2(ch2);
>> imshow(abs(ifft2(chi1)))
>> figure, imshow(abs(ifft2(chi2)))
```

MATLAB/Octave

As with ideal and Butterworth filters, the wider the high pass filter, the more of the transform we are reducing, and the less of the original image will appear in the result.

7.9 Homomorphic Filtering

If we have an image that suffers from variable illumination (dark in some sections, light in others), we may wish to enhance the contrast locally, and in particular to enhance the dark regions. Such an image may be obtained if we are recording a scene with high intensity range (say, an outdoor scene on a sunny day which

includes shadows) onto a medium with a smaller intensity range. The resulting image will contain very bright regions (those well illuminated); on the other hand, regions in shadow may appear very dark indeed.

In such a case histogram equalization won't be of much help, as the image already has high contrast. What we need to do is reduce the intensity range and at the same time increase the local contrast. We first note that the intensity of an object in an image may be considered to be a combination of two factors: the amount of light falling on it, and how much light is reflected by the object. In fact, if $f(x, y)$ is the intensity of a pixel at position (x, y) in our image, we may write:

$$f(x, y) = i(x, y)r(x, y)$$

where $i(x, y)$ is the *illumination* and $r(x, y)$ is the *reflectance*. These satisfy

$$0 < i(x, y) < \infty$$

and

$$0 < r(x, y) < 1.$$

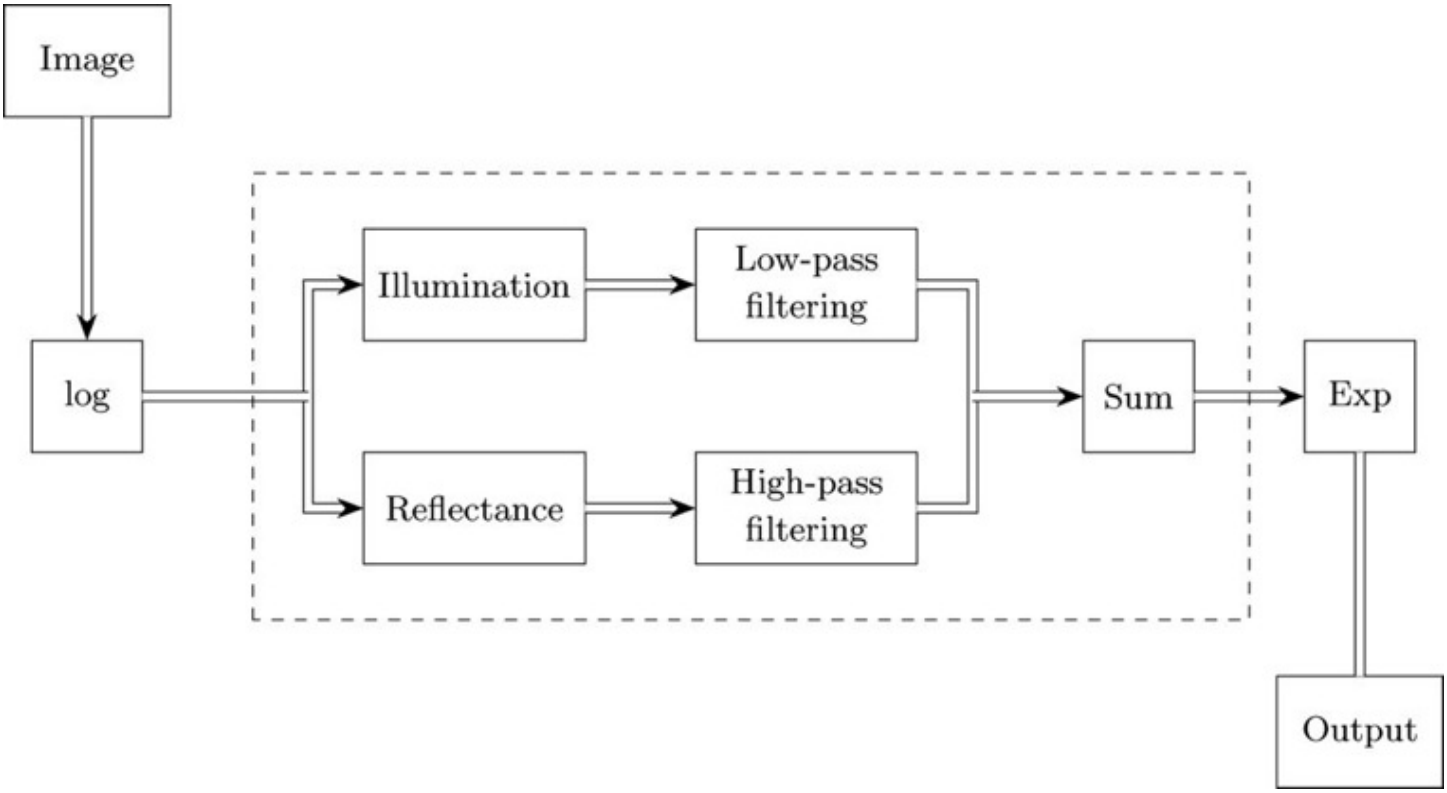
There is no (theoretical) limit as to the amount of light that can fall on an object, but reflectance is strictly bounded.

To reduce the intensity range, we need to reduce the illumination, and to increase the local contrast, we need to increase the reflectance. However, this means we need to separate $i(x, y)$ and $r(x, y)$. We can't do this directly, as the image is formed from their product. However, if we take the logarithm of the image:

$$\log f(x, y) = \log i(x, y) + \log r(x, y)$$

we can then separate the logarithms of $i(x, y)$ and $r(x, y)$. The basis of homomorphic filtering is this working with the logarithm of the image, rather than with the image directly. A schema for homomorphic filtering is given in Figure 7.26.

Figure 7.26: A schema for homomorphic filtering

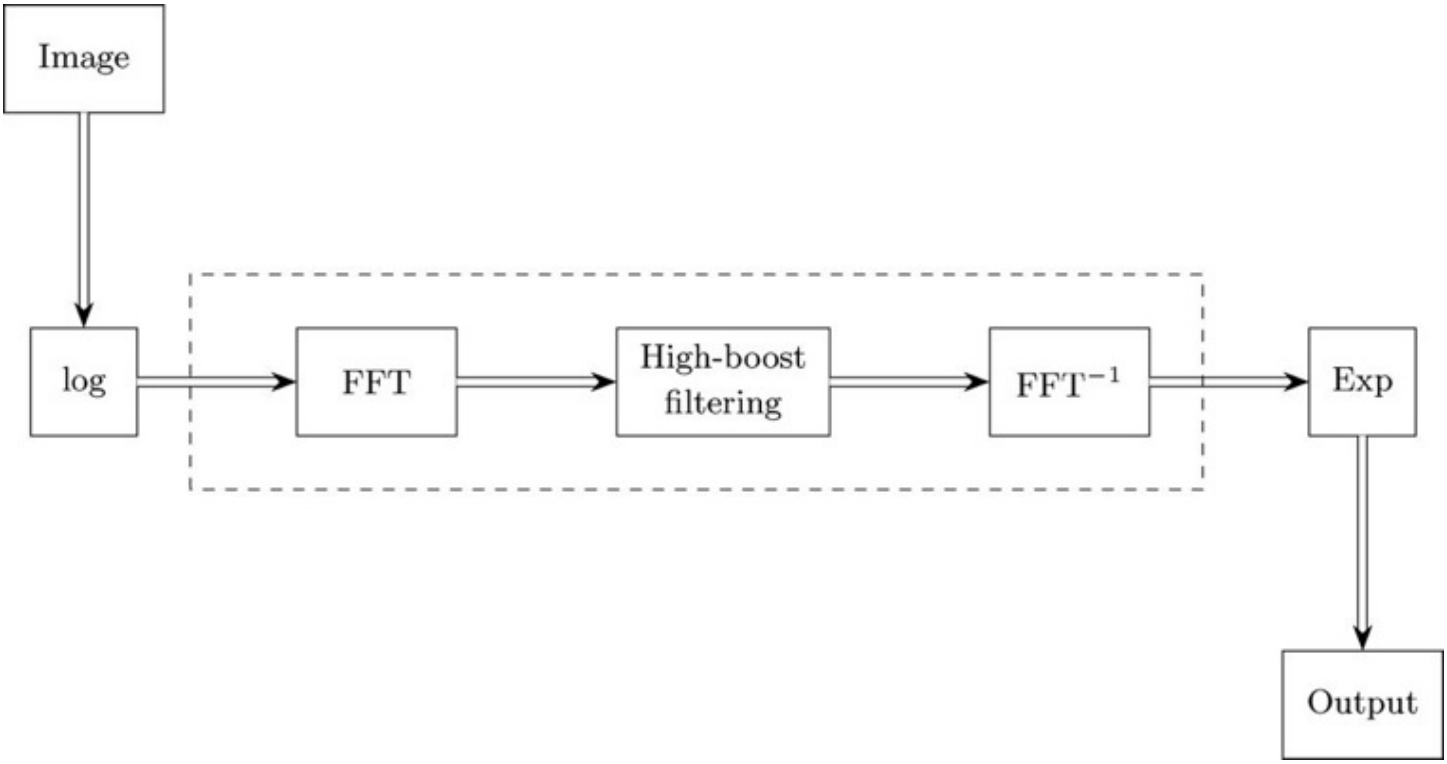


The “exp” in the second to last box simply reverses the effect of the original logarithm. We assume that the logarithm of illumination will vary slowly, and the logarithm of reflectance will vary quickly, so that the filtering processes given in Figure 7.26 will have the desired effects.

Clearly this schema will be unworkable in practice: given a value $\log f(x, y)$ we can't determine the values of its summands $\log i(x, y)$ and $\log r(x, y)$. A simpler way is to replace the dashed box in Figure 7.26 with a high boost filter of the Fourier transform. This gives the simpler schema shown in Figure 7.27.

A simple function `homfilt` to apply homomorphic filtering to an image (using a Butterworth high-boost filter) is given at the end of the chapter.

Figure 7.27: A simpler schema for homomorphic filtering



To see this in action, suppose we take an image of type `double` (so that its pixel values are between 0.0 and 1.0), and multiply it by a trigonometric function, scaled to between 0.1 and 1. If, for example, we use sin x, then the function

$$y = 0.5 + 0.4 \sin x$$

satisfies $0.1 \leq y \leq 1.0$. The result will be an image with varying illumination.

Suppose we take an image of type `double` and of size 256×256. Then we can superimpose a sine function by multiplying the image with the function above. For example:

```
>> n = im2double(imread('newborn.png'));
>> [x,y] = meshgrid(1:256,1:256);
>> nx = n.*(0.5+0.4*sin((1.3*x+0.7*y-50)/16));
```

MATLAB/Octave

or alternatively:

```
In: n = sk.util.img_as_float(io.imread('newborn.png'))
In: ar = range(256)
In: x,y = meshgrid(ar,ar)
In: nx = n*(0.5+0.4*sin((1.3*x+0.7*y-50)/16))
```

Python

The parameters used to construct the new image were chosen by trial and error to produce bands of suitable width, and on such places as to obscure detail in our image. If the original image is in Figure 7.28(a), then the result of this is shown in Figure 7.28(b).

If we apply homomorphic filtering with

```
>> xh = homfilt(x,10,2,0.5,2);
>> imshow(xh/16)
```

MATLAB/Octave

or with

Figure 7.28: Varying illumination across an image

(a) The "newborn" image

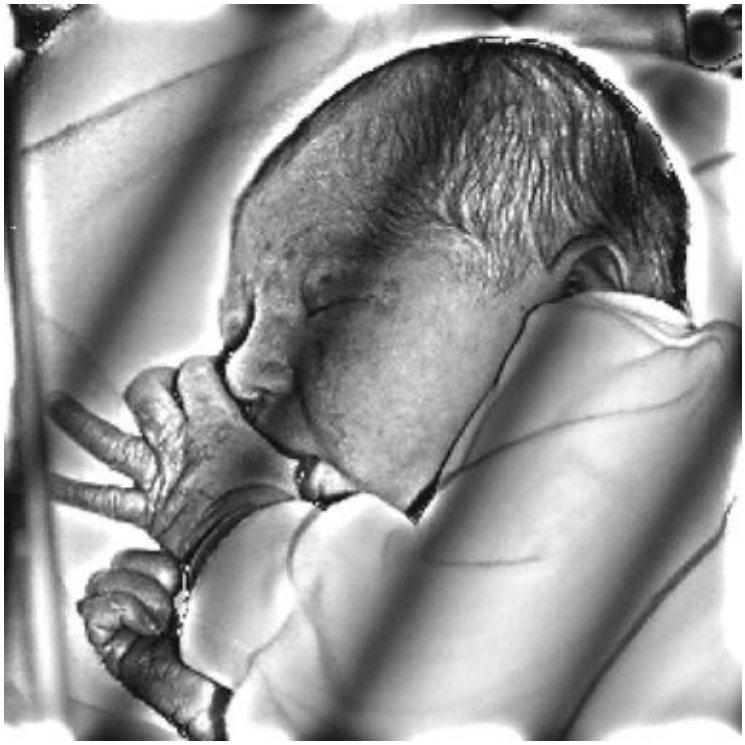
(b) Altering the illumination with a sine function

```
In :  xh = homfilt(x,10,2,0.5,2)
In :  io.imshow(xh,vmax=16)
```

Python

the result is shown in Figure 7.29.

Figure 7.29: The result of homomorphic filtering



The result shows that detail originally unobservable in the original image—especially in regions of poor illumination—is now clear. Even though the dark bands have not been completely removed, they do not obscure underlying detail as they originally did.

Figure 7.30(a) shows a picture of a ruined archway; owing to the bright light through the archway, much of the details are too dark to be seen clearly. Given this image `x`, we can perform homomorphic filtering and display the result with:

```
>> xh = homfilt(x,128,2,0.5,2);
>> imshow(xh/14)
```

MATLAB/Octave

In Python, the same `homfilt` call can be used, with the display given by

```
io.imshow(ah,vmax=12)
```

Python

and this is shown in Figure 7.30(b).

Figure 7.30: Applying homomorphic filtering to an image

(a) The arch (b) After homomorphic filtering

Note that the arch details are now much clearer, and it is even possible to make out the figure of a person standing at its base.

7.10 Programs

The `fftshow` program for viewing Fourier spectra, in MATLAB or Octave:

```
function fftshow(f,type)

% Usage:  FFTSHOW(F,TYPE)
%
% Displays the fft matrix F using imshow, where TYPE must be one of
% 'abs' or 'log'.  If TYPE='abs', then then abs(f) is displayed; if
% TYPE='log' then log(1+abs(f)) is displayed.  If TYPE is omitted, then
% 'log' is chosen as a default.
%
%  Example:
%    c=imread('cameraman.tif');
%    cf=fftshift(fft2(c));
%    fftshow(cf,'abs')
%
```

MATLAB/Octave

```
if nargin<2,
  type='log';
end

if (type=='log')
  fl = log(1+abs(f));
  fm = max(fl(:));
  imshow(im2uint8(fl/fm))
elseif (type=='abs')
  fa=abs(f);
  fm=max(fa(:));
  imshow(fa/fm)
else
  error('TYPE must be abs or log.');
end;
```

MATLAB/Octave

And here is homfilt first for MATLAB or Octave:

```
function res=homfilt(im,cutoff,order,lowgain,highgain)

% HOMFILT(IMAGE,FILTER) applies homomorphic filtering to the image IMAGE
% with the given parameters

u = im2uint8(im);

height = size(im,1);
width = size(im,2);
[x,y] = meshgrid(-floor(width/2):floor((width-1)/2),...
                 -floor(height/2):floor((height-1)/2));
lbutter = 1./(1+(sqrt(2)-1)*((x.^2+y.^2)/cutoff^2).^order);

u(find(u==0)) = 1;
lg = log(double(u));
ft = fftshift(fft2(lg));
hboost = lowgain+(highgain-lowgain)*(1-lbutter);
b = hboost.*ft;
ib = abs(ifft2(b));
res = exp(ib);
end
```

MATLAB/Octave

Moving to Python, first is fftshow:

```
#
# Read this into python with: execfile('fftshow.py') OR: import fftshow
#
#

def fftshow(im,type='log'):
    if type == 'log':
        fl = log(1+abs(im))
        fm = fl.max()
```

Python

```
        io.imshow(fl/fm)
    elif type == 'abs':
        fa = abs(im)
        fm = fa.max()
        io.imshow(fa/fm)
    else:
        print("Error: type must be abs or log")
```

Python

and homfilt:

```
def homfilt(im,cutoff,order,lowgain,highgain):
    from scipy.fftpack import fft2,ifft2,fftshift
    u = im.astype('uint8')
    u[np.where(u==0)] = 1
    lg = np.log(u.astype(float))
    ft = fftshift(fft2(lg))
    rows,cols = im.shape
    rr = arange(-(rows//2),(rows+1)//2,1.0)
    cr = arange(-(cols//2),(cols+1)//2,1.0)
    y,x = np.meshgrid(cr,rr)
    bl = 1.0/(1.0+0.414*((x*x+y*y)/cutoff**2)**order)
    f = lowgain+(highgain-lowgain)*(1.0-bl)
    b = f*ft
    ib = abs(ifft2(b))
    return np.exp(ib)
```

Python

Exercises

1. By hand, compute the DFT of each of the following sequences:
    (a) [2, 3, 4, 5]
    (b) [2, −3, 4, −5]
    (c) [−9, −8, −7, −6]

(d) [−9, 8, −7, 6]

Compare your answers with those given by your system's `fft` function.

2. For each of the transforms you computed in the previous question, compute the inverse transform by hand.

3. By hand, verify the convolution theorem for each of the following pairs of sequences:

(a) [2, 4, 6, 8] and [−1, 2 −3, 4]

(b) [4, 5, 6, 7] and [3, 1 5, −1]

4. Using your computer system, verify the convolution theorem for the following pairs of sequences:

(a) [2, −3, 5, 6, −2, −1, 3, 7] and [−1, 5, 6, 4, −3, −5, 1, 2]

(b) [7, 6, 5, 4, −4, −5, −6, −7] and [2, 2, −5, −5, 6, 6, −7, −7]

$$[2, \quad 2, \quad -5, \quad -5, \quad 6, \quad 6, \quad -7, \quad -7]$$

- 5. Consider the following matrix:

$$\begin{bmatrix} 4 & 5 & -9 & -5 \\ 3 & -7 & 1 & 2 \\ 6 & -1 & -6 & 1 \\ 3 & -1 & 7 & -5 \end{bmatrix}$$

Using your system, calculate the DFT of each row. You can do this with the commands:

```
>> a=[4 5 -9 -5;3 -7 1 2;6 -1 -6 1;3 -1 7 -5];
>> a1=fft(a')'
```

MATLAB/Octave

or

```
In :  a = np.array([[4,5,-9,-5],[3,-7,1,2],[6,-1,-6,1],[3,-1,7,-5]])
In :  fft(a)
```

Python

(The `fft` function, applied to a matrix, produces the individual DFTs of all the columns in MATLAB or Octave, and the DFTs of all the rows in Python. So for MATLAB or Octave we transpose first, so that the rows become columns, then transpose back afterward.) Now use similar commands to calculate the DFT of each column of `a1`. Compare the result with the output of the command `fft2(a)`.

- 6. Perform similar calculations as in the previous question with the matrices produced by the commands `magic(4)` and `hilb(6)`.
- 7. How do you think filtering with an averaging filter will effect the output of a Fourier Transform? Compare the DFTs of the cameraman image, and of the image after filtering with a 5 × 5 averaging filter. Can you account for the result? What happens if the averaging filter increases in size?
- 8. What is the result of two DFTs performed in succession? Apply a DFT to an image, and then another DFT to the result. Can you account for what you see?

- 9. Open up the image `engineer.png`. Experiment with applying the Fourier Transform to this image and the following filters:

    (a) Ideal filters (both low and high pass)
    (b) Butterworth filters
    (c) Gaussian filters

What is the smallest radius of a low pass ideal filter for which the face is still recognizable?

- 10. If you have access to a digital camera, or a scanner, produce a digital image of the face of somebody you know, and perform the same calculations as in the previous question.

8 Image Restoration

8.1 Introduction

Image restoration concerns the removal or reduction of degradations that have occurred during the acquisition of the image. Such degradations may include *noise*, which are errors in the pixel values, or optical effects such as out of focus blurring, or blurring due to camera motion. We shall see that some restoration techniques can be performed very successfully using neighborhood operations, while others require the use of frequency domain processes. Image restoration remains one of the most important areas of image processing, but in this chapter the emphasis will be on the techniques for dealing with restoration, rather than with the degradations themselves, or the properties of electronic equipment that give rise to image degradation.

A Model of Image Degradation

In the spatial domain, we might have an image $f(x, y)$, and a spatial filter $h(x, y)$ for which convolution with the image results in some form of degradation. For example, if $h(x, y)$ consists of a single line of ones, the result of the convolution will be a motion blur in the direction of the line. Thus, we may write

$$g(x, y) = f(x, y) * h(x, y)$$

for the degraded image, where the symbol * represents spatial filtering. However, this is not all. We must consider noise, which can be modeled as an additive function to the convolution. Thus, if $n(x, y)$ represents random errors which may occur, we have as our degraded image:

$$g(x, y) = f(x, y) * h(x, y) + n(x, y).$$

We can perform the same operations in the frequency domain, where convolution is replaced by multiplication, and addition remains as addition because of the linearity of the Fourier transform. Thus,

$$G(i, j) = F(i, j)H(i, j) + N(i, j)$$

represents a general image degradation, where of course F, H, and N are the Fourier transforms of f, h, and n, respectively.

If we knew the values of H and N we could recover F by writing the above equation as